Part V

Sorting and Searching

Chapter 13

Divide and conquer, Quick sort vs. Merge sort

13.1 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.

13.2 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;

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.

Figure 13.1: Instruct kids to stand in a line

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.

13.2.1 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.

$$sort(L) = \begin{cases} \Phi & : \quad L = \Phi \\ sort(\{x|x \in L', x \leq l_1\}) \cup \{l_1\} \cup sort(\{x|x \in L', l_1 < x\}) & : \quad otherwise \end{cases} \tag{13.1}$$

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 $p_1, p_2, ...$ ZF expression is originally used for representing *set*, we extend it to express list for the sake of brevity. There can be duplicated elements, and different permutations represent for different 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
```

13.2.2 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.

13.2.3 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.

$$partition(p, L) = \begin{cases} (\Phi, \Phi) & : & L = \Phi \\ (\{l_1\} \cup A, B) & : & p(l_1), (A, B) = partition(p, L') \\ (A, \{l_1\} \cup B) & : & \neg p(l_1) \end{cases} \tag{13.2}$$

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

$$sort(L) = \begin{cases} \Phi & : & L = \Phi \\ sort(A) \cup \{l_1\} \cup sort(B) & : & otherwise, (A, B) = partition(\lambda_x x \leq l_1, L') \end{cases} \tag{13.3}$$

Translating this new algorithm into Haskell yields the below code.

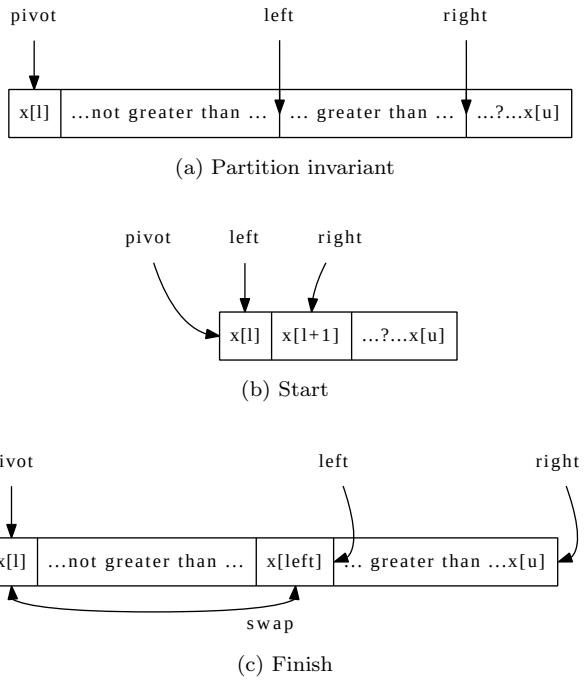
```
sort [] = []
sort (x:xs) = sort as ++ [x] ++ sort bs where
    (as, bs) = partition (≤ x) xs

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.

Figure 13.2 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 13.2 (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) Partition invariant

(b) Start

(c) Finish

Figure 13.2: Partition a range of array by using the left most element as pivot.

- 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 13.2 (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 13.2 (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].

1: **function** PARTITION(A, l, u)
2: $p \leftarrow A[l]$ ▷ the pivot
3: $L \leftarrow l$ ▷ the left mark
4: **for** $R \in [l+1, u]$ **do** ▷ iterate on the right mark
5: **if** $\neg(p < A[R])$ **then** ▷ negate of $<$ is enough for strict weak order
6: $L \leftarrow L + 1$
7: EXCHANGE $A[L] \leftrightarrow A[R]$
8: EXCHANGE $A[L] \leftrightarrow p$
9: **return** $L + 1$ ▷ The partition position

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

(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

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.

[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.

```
1: procedure QUICK-SORT(A, l, u)
2:     if l < u then
3:         m ← PARTITION(A, l, u)
4:         QUICK-SORT(A, l, m − 1)
5:         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);
    }
}
```

13.2.4 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.

$$partition(p, L) = fold(f(p), (\Phi, \Phi), L) \tag{13.4}$$

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.

$$f(p, x, (A, B)) = \begin{cases} (\{x\} \cup A, B) & : \quad p(x) \\ (A, \{x\} \cup B) & : \quad otherwise(\neg p(x)) \end{cases} \tag{13.5}$$

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)
```

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).

$$partition(p, L, A, B) = \begin{cases} (A, B) & : & L = \Phi \\ partition(p, L', \{l_1\} \cup A, B) & : & p(l_1) \\ partition(p, L', A, \{l_1\} \cup B) & : & otherwise \end{cases} \tag{13.6}$$

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.

$$sort(L) = \begin{cases} \Phi & : & L = \Phi \\ sort(A) \cup \{l_1\} \cup sort(B) & : & otherwise \end{cases} \tag{13.7}$$

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

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

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:

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

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.

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

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.

$$sort'(L, S) = \begin{cases} S & : & L = \Phi \\ sort(A, \{l_1\} \cup sort(B, S)) & : & otherwise \end{cases} \tag{13.8}$$

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 13.1

- 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.

13.3 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 13.3.

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.

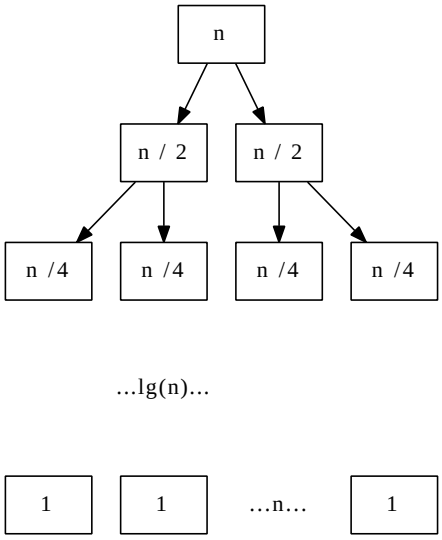


Figure 13.3: In the best case, quick sort divides the sequence into two slices with same length.

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.

13.3.1 Average case analysis ⋆

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:

$$P(i, j) = \frac{2}{j - i + 1} \tag{13.9}$$

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

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(i, j) \tag{13.10}$$

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:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} \end{aligned} \tag{13.11}$$

Using the harmonic series [18]

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

$$C(n) = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n) \tag{13.12}$$

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.

$$T(n) = T(i) + T(n - i - 1) + cn \tag{13.13}$$

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:

$$\begin{aligned} T(n) &= E(T(i)) + E(T(n-i-1)) + cn \\ &= \frac{1}{n}\sum_{i=0}^{n-1} T(i) + \frac{1}{n}\sum_{i=0}^{n-1} T(n-i-1) + cn \\ &= \frac{1}{n}\sum_{i=0}^{n-1} T(i) + \frac{1}{n}\sum_{j=0}^{n-1} T(j) + cn \\ &= \frac{2}{n}\sum_{i=0}^{b-1} T(i) + cn \end{aligned} \tag{13.14}$$

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

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + cn^2 \tag{13.15}$$

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

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + c(n-1)^2 \tag{13.16}$$

Subtract equation (13.15) and (13.16) can eliminate all the $T(i)$ for $0 \leq i < n-1$.

$$nT(n) = (n+1)T(n-1) + 2cn - c \tag{13.17}$$

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

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} \tag{13.18}$$

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

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

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

$$\cdots$$

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

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

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{k=3}^{n+1} \frac{1}{k} \tag{13.19}$$

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

$$O(\frac{T(n)}{n+1}) = O(\frac{T(1)}{2} + 2c \ln n + \gamma + \epsilon_n) = O(\lg n) \tag{13.20}$$

Thus

$$O(T(n)) = O(n \lg n) \tag{13.21}$$

Exercise 13.2

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

13.4 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.

13.4.1 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.

$$sort(L) = \begin{cases} \Phi & : & L = \Phi \\ sort(S) \cup sort(E) \cup sort(G) & : & otherwise \end{cases} \tag{13.22}$$

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

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

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.

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

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:

$$sort'(L, A) = \begin{cases} A & : & L = \Phi \\ sort(S, E \cup sort(G, A)) & : & otherwise \end{cases} \tag{13.23}$$

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.

$$partition(p, L, S, E, G) = \begin{cases} (S, E, G) & : & L = \Phi \\ partition(p, L', \{l_1\} \cup S, E, G) & : & l_1 < p \\ partition(p, L', S, \{l_1\} \cup E, G) & : & l_1 = p \\ partition(p, L', S, E, \{l_1\} \cup G) & : & p < l_1 \end{cases} \tag{13.24}$$

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 [1], 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
```

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.

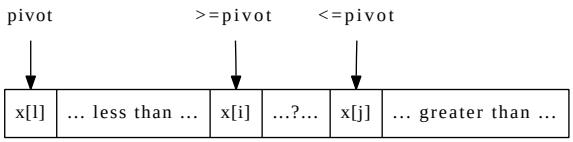
At this time, all elements before the left pointer i are strictly less than the pivot, while all elements after the right pointer j are greater than the pivot. i points to an element which is either greater than or equal to the pivot; while j points to an element which is either less than or equal to the pivot, the situation at this stage is illustrated in figure 13.4 (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.

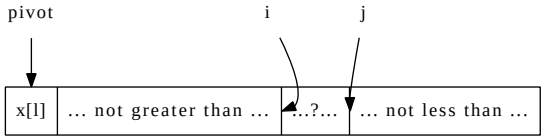
At any time point during partition. There is invariant that all elements before i (including the one pointed by i) are not greater than the pivot; while all elements after j (including the one pointed by j) are not less than the pivot. The elements between i and j haven't been examined yet. This invariant is shown in figure 13.4 (b).

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

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



(a) When pointer i, and j stop



(b) Partition invariant

Figure 13.4: Partition a range of array by using the left most element as the pivot.

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.

```
1: procedure SORT(A, l, u)                                  ▷ sort range [l, u)
2:     if u − l > 1 then                  ▷ More than 1 element for non-trivial case
3:         i ← l, j ← u
4:         pivot ← A[l]
5:         loop
6:             repeat
7:                 i ← i + 1
8:             until A[i] ≥ pivot     ▷ Need handle error case that i ≥ u in fact.
9:             repeat
10:                j ← j − 1
11:            until A[j] ≤ pivot     ▷ Need handle error case that j < l in fact.
12:            if j < i then
13:                break
14:            EXCHANGE A[i] ↔ A[j]
15:        EXCHANGE A[l] ↔ A[j]                                  ▷ Move the pivot
16:        SORT(A, l, j)
17:        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) {
```

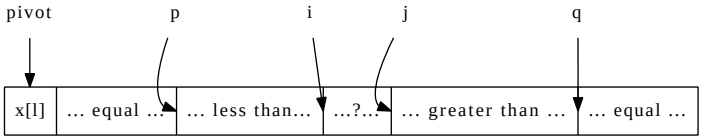
```
            while (i < u && xs[++i] < xs[pivot]);
            while (j >= l && xs[pivot] < xs[--j]);
            if (j < i) break;
            swap(xs[i], xs[j]);
        }
        swap(xs[pivot], xs[j]);
        qsort(xs, l, j);
        qsort(xs, i, u);
    }
}
```

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.

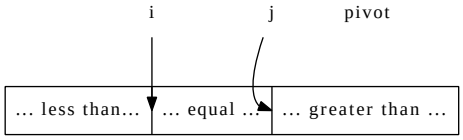
3-way partition

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 13.5 (a) [5] [6].



(a) Invariant of 3-way partition



(b) Swapping the equal parts to the middle

Figure 13.5: 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 13.5 (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.

```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
3:         i ← l, j ← u
4:         p ← l, q ← u                ▷ points to the boundaries for equal elements
5:         pivot ← A[l]
6:         loop
7:             repeat
8:                 i ← i + 1
9:             until A[i] ≥ pivot                ▷ Skip the error handling for i ≥ u
10:            repeat
11:                j ← j − 1
12:            until A[j] ≤ pivot                ▷ Skip the error handling for j < l
13:            if j ≤ i then
14:                break          ▷ Note the difference form the above algorithm
15:            EXCHANGE A[i] ↔ A[j]
16:            if A[i] = pivot then                ▷ Handle the equal elements
17:                p ← p + 1
18:                EXCHANGE A[p] ↔ A[i]
19:            if A[j] = pivot then
20:                q ← q − 1
21:                EXCHANGE A[q] ↔ A[j]
22:        if i = j ∧ A[i] = pivot then                ▷ A special case
23:            j ← j − 1, i ← i + 1
24:        for k from l to p do   ▷ Swap the equal elements to the middle part
25:            EXCHANGE A[k] ↔ A[j]
26:            j ← j − 1
27:        for k from u − 1 down-to q do
28:            EXCHANGE A[k] ↔ A[i]
29:            i ← i + 1
30:        SORT(A, l, j + 1)
31:        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 13.6, 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.

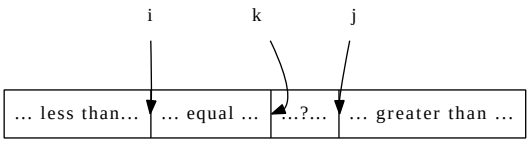


Figure 13.6: 3-way partition based on N. Lumoto's method.

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.

```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
3:         i ← l, j ← u, k ← l + 1
4:         pivot ← A[i]
5:         while k < j do
```

```
6:          while pivot < A[k] do
7:              j ← j − 1
8:              Exchange A[k] ↔ A[j]
9:          if A[k] < pivot then
10:             Exchange A[k] ↔ A[i]
11:             i ← i + 1
12:         k ← k + 1
13:     Sort(A, l, i)
14:     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 13.3

- 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.

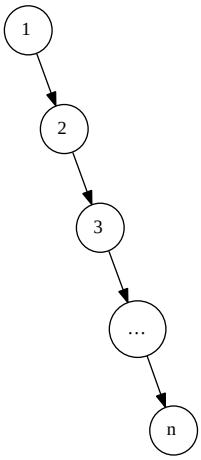
13.5 Engineering solution to the worst case

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.

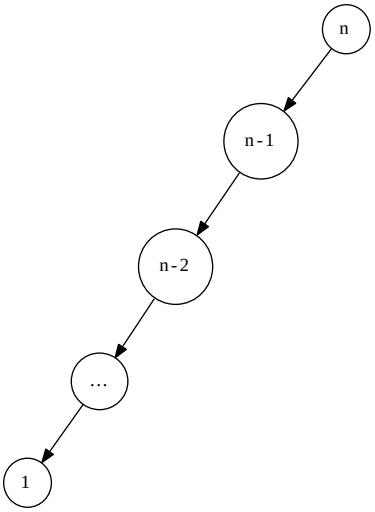
Consider the two extreme cases, $\{x_1 < x_2 < ... < x_n\}$ and $\{y_1 > y_2 > ... > y_n\}$. The partition results are shown in figure 13.7.

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 13.8.

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,

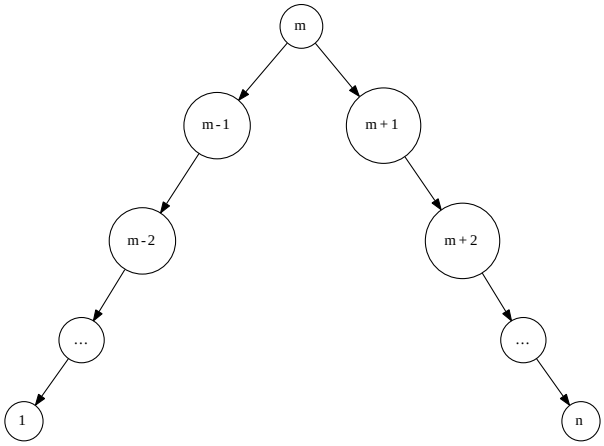


(a) The partition tree for $\{x_1 < x_2 < ... < x_n\}$, There aren't any elements less than or equal to the pivot (the first element) in every partition.

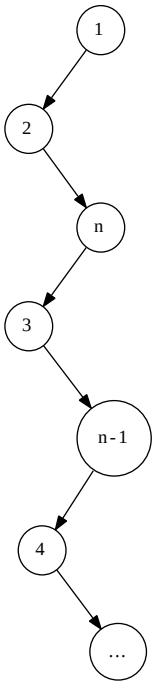


(b) The partition tree for $\{y_1 > y_2 > ... > y_n\}$, There aren't any elements greater than or equal to the pivot (the first element) in every partition.

Figure 13.7: The two worst cases.



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



(b) A zig-zag partition tree.

Figure 13.8: Another two worst cases.

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.

```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
3:         m ← ⌊(l+u)/2⌋                    ▷ Need handle overflow error in practice
4:         if A[m] < A[l] then                                  ▷ Ensure A[l] ≤ A[m]
5:             EXCHANGE A[l] ↔ A[r]
6:         if A[u − 1] < A[l] then                              ▷ Ensure A[l] ≤ A[u − 1]
7:             EXCHANGE A[l] ↔ A[u − 1]
8:         if A[u − 1] < A[m] then                              ▷ Ensure A[m] ≤ A[u − 1]
9:             EXCHANGE A[m] ↔ A[u]
10:        EXCHANGE A[l] ↔ A[m]
11:        (i, j) ← PARTITION(A, l, u)
12:        SORT(A, l, i)
13:        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.

```
1: procedure SORT(A, l, u)
2:     if u − l > 1 then
```

```
3:        EXCHANGE A[l] ↔ A[ RANDOM(l, u) ]
4:        (i, j) ← PARTITION(A, l, u)
5:        SORT(A, l, i)
6:        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.

13.6 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.

```
1: procedure SORT(A, l, u)
2:     if u − l > CUT-OFF then
3:         QUICK-SORT(A, l, u)
4:     else
5:         INSERTION-SORT(A, l, u)
```

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

Exercise 13.4

- 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.

13.7 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 [7].

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

$$unfold(L) = \begin{cases} \Phi & : & L = \Phi \\ tree(T_l, l_1, T_r) & : & otherwise \end{cases} \tag{13.25}$$

Where

$$\begin{aligned} T_l &= unfold(\{a | a \in L', a \leq l_1\}) \\ T_r &= unfold(\{a | a \in L', l_1 < a\}) \end{aligned} \tag{13.26}$$

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:

$$toList(T) = \begin{cases} \Phi & : & T = \Phi \\ toList(left(T)) \cup \{key(T)\} \cup toList(right(T)) & : & otherwise \end{cases} \tag{13.27}$$

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

$$quickSort = toList \cdot unfold \tag{13.28}$$

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].

13.8 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.

13.8.1 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.

$$sort(L) = \begin{cases} \Phi & : & L = \Phi \\ merge(sort(L_1), sort(L_2)) & : & otherwise, (L_1, L_2) = splitAt(\lfloor \frac{|L|}{2} \rfloor, L) \end{cases} \tag{13.29}$$

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:

$$splitAt(n, L) = \begin{cases} (\Phi, L) & : & n = 0 \\ (\{l_1\} \cup A, B) & : & otherwise, (A, B) = splitAt(n - 1, L') \end{cases} \tag{13.30}$$

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 13.9. 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.

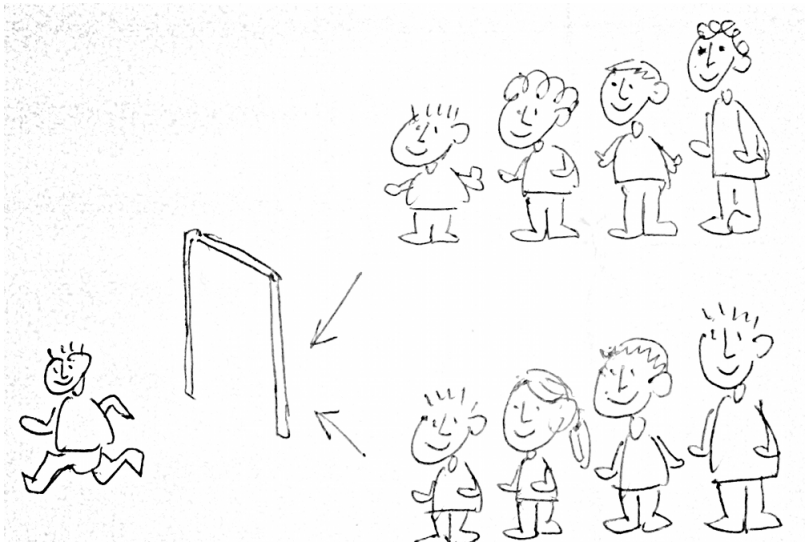


Figure 13.9: Two lines of kids pass a door.

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.

$$merge(A, B) = \begin{cases} A & : & B = \Phi \\ B & : & A = \Phi \\ \{a_1\} \cup merge(A', B) & : & a_1 \leq b_1 \\ \{b_1\} \cup merge(A, B') & : & otherwise \end{cases} \tag{13.31}$$

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.

```
1: procedure SORT(A)
2:     if |A| > 1 then
3:         m ← ⌊|A|/2⌋
4:         X ← COPY-ARRAY(A[1...m])
5:         Y ← COPY-ARRAY(A[m + 1...|A|])
6:         SORT(X)
7:         SORT(Y)
8:         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.

```
1: procedure MERGE(A, X, Y)
2:     i ← 1, j ← 1, k ← 1
3:     m ← |X|, n ← |Y|
4:     while i ≤ m ∧ j ≤ n do
5:         if X[i] < Y[j] then
6:             A[k] ← X[i]
7:             i ← i + 1
8:         else
9:             A[k] ← Y[j]
10:            j ← j + 1
11:        k ← k + 1
12:    while i ≤ m do
13:        A[k] ← X[i]
14:        k ← k + 1
15:        i ← i + 1
16:    while j ≤ n do
17:        A[k] ← Y[j]
18:        k ← k + 1
19:        j ← 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
```

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 13.3. 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.

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + T(\frac{n}{2}) + cn \\ &= 2T(\frac{n}{2}) + cn \end{aligned} \tag{13.32}$$

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.

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 13.10 illustrates this idea.

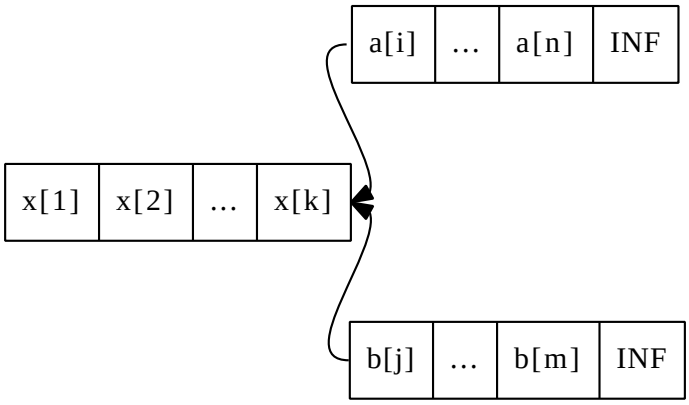


Figure 13.10: Merge with ∞ as sentinels.

```
1:  procedure MERGE(A, X, Y)
2:      APPEND(X, ∞)
3:      APPEND(Y, ∞)
4:      i ← 1, j ← 1
5:      for k ← from 1 to |A| do
6:          if X[i] < Y[j] then
7:              A[k] ← X[i]
8:              i ← i + 1
9:          else
10:             A[k] ← Y[j]
11:             j ← 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

[3]The complex effects caused by lazy evaluation is ignored here, please refer to [7] for detail

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

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.

```
1: procedure SORT(A)
2:     B ← CREATE-ARRAY(|A|)
3:     SORT'(A, B, 1, |A|)

4: procedure SORT'(A, B, l, u)
5:     if u − l > 0 then
6:         m ← ⌊(l+u)/2⌋
7:         SORT'(A, B, l, m)
8:         SORT'(A, B, m + 1, u)
9:         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.

```
1: procedure MERGE'(A, B, l, m, u)
2:     i ← l, j ← m + 1, k ← l
3:     while i ≤ m ∧ j ≤ u do
4:         if A[i] < A[j] then
```

```
5:              B[k] ← A[i]
6:              i ← i + 1
7:          else
8:              B[k] ← A[j]
9:              j ← j + 1
10:         k ← k + 1
11:     while i ≤ m do
12:         B[k] ← A[i]
13:         k ← k + 1
14:         i ← i + 1
15:     while j ≤ u do
16:         B[k] ← A[j]
17:         k ← k + 1
18:         j ← j + 1
19:     for i ← from l to u do                    ▷ Copy back
20:         A[i] ← 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.

$$split(L) = \begin{cases} (\Phi, \Phi) & : & L = \Phi \\ (\{l_1\}, \Phi) & : & |L| = 1 \\ (\{l_1\} \cup A, \{l_2\} \cup B) & : & otherwise, (A, B) = split(L'') \end{cases} \tag{13.33}$$

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
```

13.9 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.

13.9.1 Naive in-place merge

The first idea is straightforward. As illustrated in figure 13.11, 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.

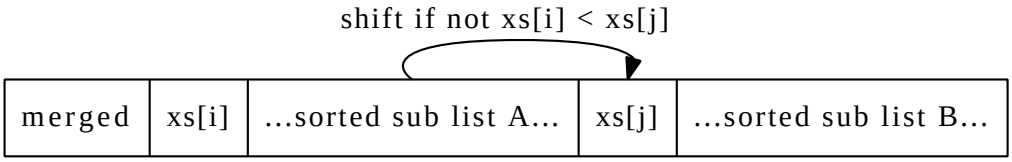


Figure 13.11: Naive in-place merge

```
1: procedure MERGE(A, l, m, u)
2:     while l ≤ m ∧ m ≤ u do
3:         if A[l] < A[m] then
4:             l ← l + 1
5:         else
6:             x ← A[m]
7:             for i ← m down-to l + 1 do                ▷ Shift
8:                 A[i] ← A[i − 1]
9:             A[l] ← 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);
    }
}
```

13.9.2 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 13.12.

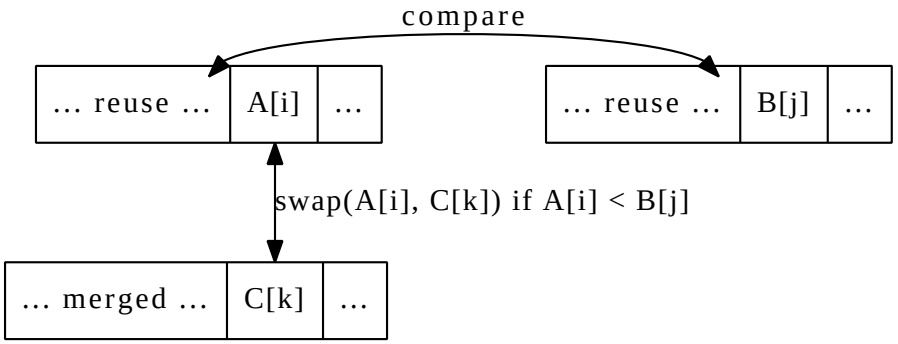


Figure 13.12: Merge without overwriting working area.

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 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.

```
1:  procedure MERGE(A, [i, m), [j, n), k)
2:      while i < m ∧ j < n do
3:          if A[i] < A[j] then
4:              EXCHANGE A[k] ↔ A[i]
5:              i ← i + 1
6:          else
7:              EXCHANGE A[k] ↔ A[j]
8:              j ← j + 1
9:          k ← k + 1
10:     while i < m do
11:         EXCHANGE A[k] ↔ A[i]
12:         i ← i + 1
13:         k ← k + 1
14:     while j < m do
15:         EXCHANGE A[k] ↔ A[j]
16:         j ← j + 1
17:         k ← 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 13.13?

Figure 13.13: 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 13.14. 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.

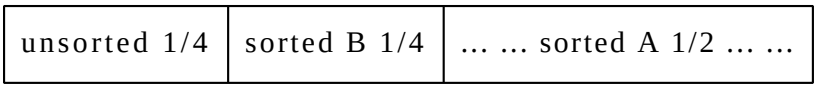


Figure 13.14: A and B must be merged at sometime.

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 13.14.

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 13.15 (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

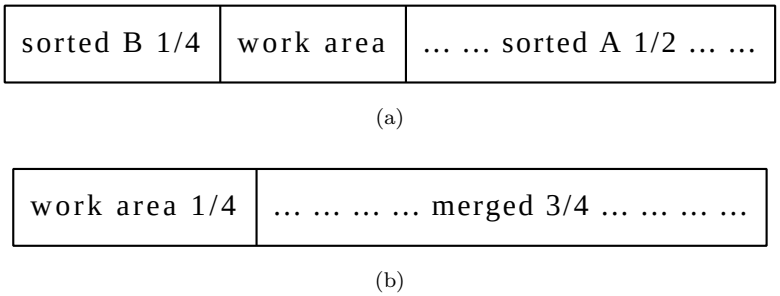


Figure 13.15: Merge A and B with the working area.

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;

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 13.15 (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.

```
1:  procedure SORT(A, l, u)
2:      if u − l > 0 then
3:          m ← ⌊(l+u)/2⌋
4:          w ← l + u − m
5:          SORT'(A, l, m, w)          ▷ The second half contains sorted elements
6:          while w − l > 1 do
7:              u' ← w
8:              w ← ⌈(l+u')/2⌉          ▷ Ensure the working area is big enough
9:              SORT'(A, w, u', l)      ▷ The first half holds the sorted elements
10:             MERGE(A, [l, l + u' − w], [u', u], w)
11:         for i ← w down-to l do                    ▷ Switch to insertion sort
12:             j ← i
13:             while j ≤ u ∧ A[j] < A[j − 1] do
14:                 EXCHANGE A[j] ↔ A[j − 1]
```

15: $j \leftarrow j + 1$

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.

```
1: procedure SORT'(A, l, u, w)
2:     if u − l > 0 then
3:         m ← ⌊(l+u)/2⌋
4:         SORT(A, l, m)
5:         SORT(A, m + 1, u)
6:         MERGE(A, [l, m], [m + 1, u], w)
7:     else                          ▷ Exchange all elements to the working area
8:         while l ≤ u do
9:             EXCHANGE A[l] ↔ A[w]
10:            l ← l + 1
11:            w ← 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.

$$T(n) = T(\frac{n}{2}) + c\frac{n}{2} + T(\frac{n}{4}) + c\frac{3n}{4} + T(\frac{n}{8}) + c\frac{7n}{8} + ... \tag{13.34}$$

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.

13.9.3 In-place merge sort vs. 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.

```
1: function SPLIT(L)
2:     (A, B) ← (Φ, Φ)
3:     while L ≠ Φ do
4:         p ← L
5:         L ← NEXT(L)
```

6: $A \leftarrow \textsc{Link}(p, A)$
7: $\textsc{Exchange}\ A \leftrightarrow B$
8: **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 13.5

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

13.10 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.

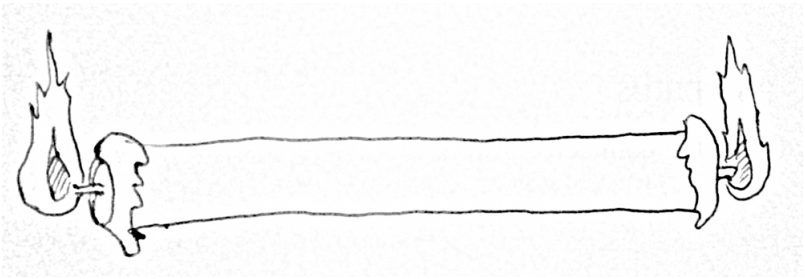


Figure 13.16: Burn a candle from both ends

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.

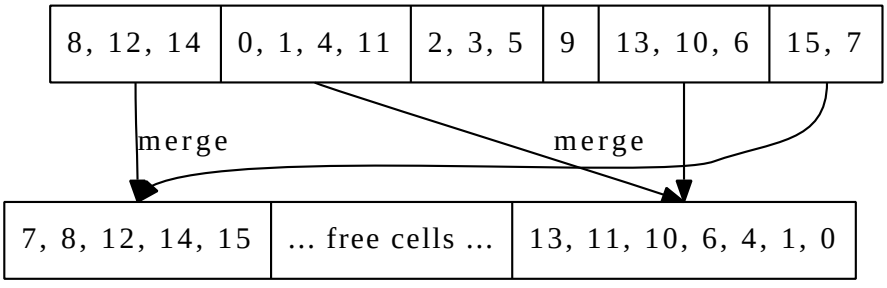


Figure 13.17: Nature merge sort

Figure 13.17 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 13.18 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.

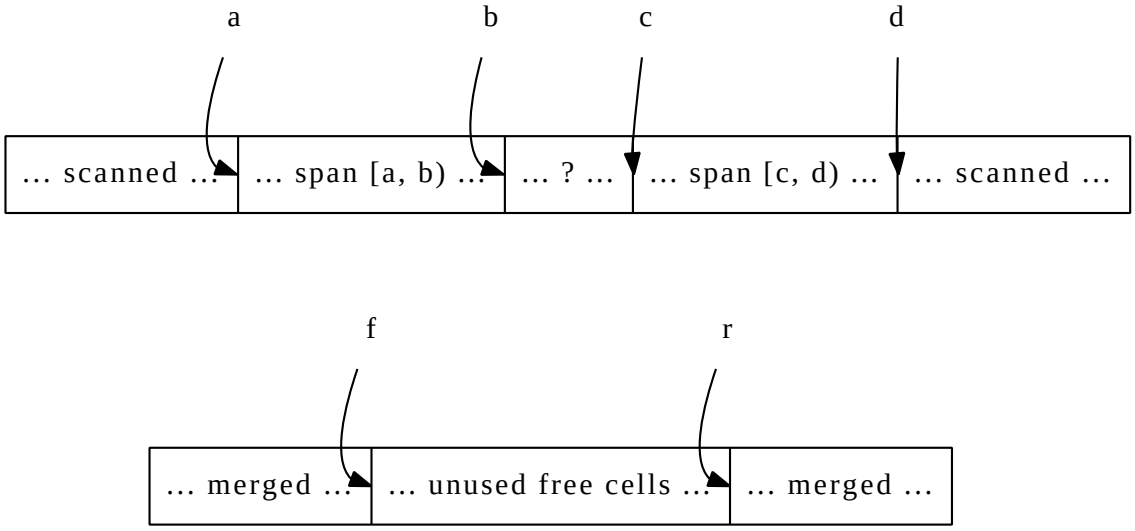


Figure 13.18: Invariant during nature merge sort

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

```
1: function SORT(A)
2:     if |A| > 1 then
3:         n ← |A|
4:         B ← CREATE-ARRAY(n)                          ▷ Create the working area
5:         loop
6:             [a, b) ← [1, 1)
7:             [c, d) ← [n + 1, n + 1)
8:             f ← 1, r ← n          ▷ front and rear pointers to the working area
9:             t ← False                                  ▷ merge to front or rear
10:            while b < c do                  ▷ There are still elements for scan
11:                repeat                                          ▷ Span [a, b)
12:                    b ← b + 1
13:                until b ≥ c ∨ A[b] < A[b − 1]
14:                repeat                                          ▷ Span [c, d)
15:                    c ← c − 1
16:                until c ≤ b ∨ A[c − 1] < A[c]
17:                if c < b then                                  ▷ Avoid overlap
18:                    c ← b
19:                if b − a ≥ n then     ▷ Done if [a, b) spans to the whole array
20:                    return A
21:                if t then                                      ▷ merge to front
22:                    f ← MERGE(A, [a, b), [c, d), B, f, 1)
23:                else                                           ▷ merge to rear
24:                    r ← MERGE(A, [a, b), [c, d), B, r, −1)
25:                a ← b, d ← c
26:                t ← ¬t                          ▷ Switch the merge direction
27:            EXCHANGE A ↔ B                                  ▷ Switch working area
28:    return A
```

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

```
1: function MERGE(A, [a, b), [c, d), B, w, Δ)
2:     while a < b ∧ c < d do
3:         if A[a] < A[d − 1] then
4:             B[w] ← A[a]
5:             a ← a + 1
6:         else
7:             B[w] ← A[d − 1]
8:             d ← d − 1
9:         w ← w + Δ
10:    while a < b do
11:        B[w] ← A[a]
12:        a ← a + 1
13:        w ← w + Δ
14:    while c < d do
15:        B[w] ← A[d − 1]
16:        d ← d − 1
17:        w ← w + Δ
```

18: **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.

$$sort(L) = sort'(group(L)) \tag{13.35}$$

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.

$$group(L) = \begin{cases} \{L\} & : & |L| \leq 1 \\ \{\{l_1\} \cup L_1, L_2, ...\} & : & l_1 \leq l_2, \{L_1, L_2, ...\} = group(L') \\ \{\{l_1\}, L_1, L_2, ...\} & : & otherwise \end{cases} \tag{13.36}$$

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
```

[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.

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.

$$sort'(\mathbb{L}) = \begin{cases} \Phi & : \mathbb{L} = \Phi \\ L_1 & : \mathbb{L} = \{L_1\} \\ sort'(mergePairs(\mathbb{L})) & : otherwise \end{cases} \tag{13.37}$$

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 $sort'$.

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

$$mergePairs(L) = \begin{cases} L & : |L| \leq 1 \\ \{merge(L_1, L_2)\} \cup mergePairs(L'') & : otherwise \end{cases} \tag{13.38}$$

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.

$$sort(L) = fold(merge, \Phi, group(L)) \tag{13.39}$$

Translate this version to Haskell yields the folding version.

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

Exercise 13.6

- 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?

13.11 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 13.19

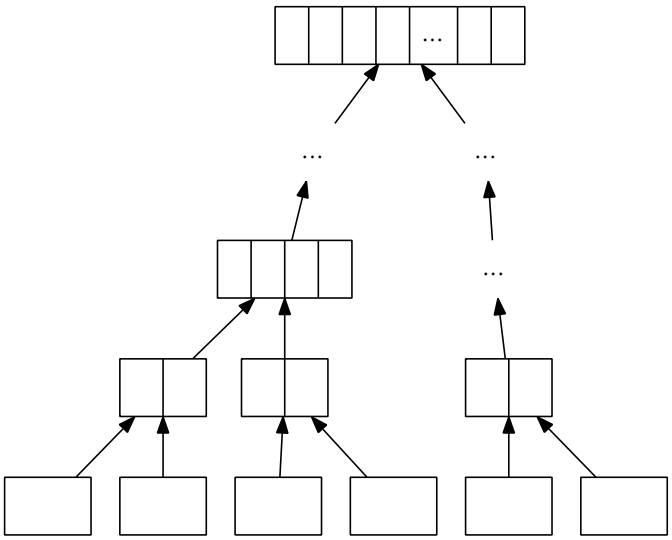


Figure 13.19: Bottom-up merge sort

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.

$$sort(L) = sort'(wraps(L)) \tag{13.40}$$

$$wraps(L) = \begin{cases} \Phi & : \quad L = \Phi \\ \{\{l_1\}\} \cup wraps(L') & : \quad otherwise \end{cases} \tag{13.41}$$

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

$$sort(L) = sort'(map(\lambda_x \cdot \{x\}, L)) \tag{13.42}$$

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 [6]. 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.

$$sort(L) = sort'(groupBy(\lambda_{x,y} \cdot False, L)) \tag{13.43}$$

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.

```
1: function SORT(A)
2:     B ← Φ
3:     for ∀a ∈ A do
4:         B ← APPEND({a})
5:     N ← |B|
6:     while N > 1 do
7:         for i ← from 1 to ⌊N/2⌋ do
8:             B[i] ← MERGE(B[2i − 1], B[2i])
9:         if ODD(N) then
10:            B[⌈N/2⌉] ← B[N]
11:        N ← ⌈N/2⌉
12:    if B = Φ then
13:        return Φ
14:    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 13.7

- 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.

13.12 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.

13.13 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].

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

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

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 vs. 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 [7].

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 13.8

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

Bibliography

[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)

Chapter 14

Searching

14.1 Introduction

Searching is quite a big and important area. Computer makes many hard searching problems realistic. They are almost impossible for human beings. A modern industry robot can even search and pick the correct gadget from the pipeline for assembly; A GPS car navigator can search among the map, for the best route to a specific place. The modern mobile phone is not only equipped with such map navigator, but it can also search for the best price for Internet shopping.

This chapter just scratches the surface of elementary searching. One good thing that computer offers is the brute-force scanning for a certain result in a large sequence. The divide and conquer search strategy will be briefed with two problems, one is to find the k-th big one among a list of unsorted elements; the other is the popular binary search among a list of sorted elements. We'll also introduce the extension of binary search for multiple-dimension data.

Text matching is also very important in our daily life, two well-known searching algorithms, Knuth-Morris-Pratt (KMP) and Boyer-Moore algorithms will be introduced. They set good examples for another searching strategy: information reusing.

Besides sequence search, some elementary methods for searching solution for some interesting problems will be introduced. They were mostly well studied in the early phase of AI (artificial intelligence), including the basic DFS (Depth first search), and BFS (Breadth first search).

Finally, Dynamic programming will be briefed for searching optimal solutions, and we'll also introduce about greedy algorithm which is applicable for some special cases.

All algorithms will be realized in both imperative and functional approaches.

14.2 Sequence search

Although modern computer offers fast speed for brute-force searching, and even if the Moore's law could be strictly followed, the grows of huge data is too fast to be handled well in this way. We've seen a vivid example in the introduction chapter of this book. It's why people study the computer search algorithms.

14.2.1 Divide and conquer search

One solution is to use divide and conquer approach. That if we can repeatedly scale down the search domain, the data being dropped needn't be examined at all. This will definitely speed up the search.

k-selection problem

Consider a problem of finding the k-th smallest one among n elements. The most straightforward idea is to find the minimum first, then drop it and find the second minimum element among the rest. Repeat this minimum finding and dropping k steps will give the k-th smallest one. Finding the minimum among n elements costs linear $O(n)$ time. Thus this method performs $O(kn)$ time, if k is much smaller than n.

Another method is to use the 'heap' data structure we've introduced. No matter what concrete heap is used, e.g. binary heap with implicit array, Fibonacci heap or others, Accessing the top element followed by popping is typically bound $O(\lg n)$ time. Thus this method, as formalized in equation (14.1) and (14.2) performs in $O(k \lg n)$ time, if k is much smaller than n.

$$top(k, L) = find(k, heapify(L)) \tag{14.1}$$

$$find(k, H) = \begin{cases} top(H) & : \quad k = 0 \\ find(k-1, pop(H)) & : \quad otherwise \end{cases} \tag{14.2}$$

However, heap adds some complexity to the solution. Is there any simple, fast method to find the k-th element?

The divide and conquer strategy can help us. If we can divide all the elements into two sub lists A and B, and ensure all the elements in A is not greater than any elements in B, we can scale down the problem by following this method[1]:

1. Compare the length of sub list A and k;
2. If $k < |A|$, the k-th smallest one must be contained in A, we can drop B and *further search* in A;
3. If $|A| < k$, the k-th smallest one must be contained in B, we can drop A and *further search* the $(k - |A|)$-th smallest one in B.

Note that the *italic font* emphasizes the fact of recursion. The ideal case always divides the list into two equally big sub lists A and B, so that we can halve the problem each time. Such ideal case leads to a performance of $O(n)$ linear time.

Thus the key problem is how to realize dividing, which collects the first m smallest elements in one sub list, and put the rest in another.

This reminds us the partition algorithm in quick sort, which moves all the elements smaller than the pivot in front of it, and moves those greater than the pivot behind it. Based on this idea, we can develop a divide and conquer k-selection algorithm, which is called quick selection algorithm.

[1] This actually demands a more accurate definition of the k-th smallest in L: It's equal to the k-the element of L', where L' is a permutation of L, and L' is in monotonic non-decreasing order.

1. Randomly select an element (the first for instance) as the pivot;
2. Moves all elements which aren't greater than the pivot in a sub list A; and moves the rest to sub list B;
3. Compare the length of A with k, if $|A| = k - 1$, then the pivot is the k-th smallest one;
4. If $|A| > k - 1$, recursively find the k-th smallest one among A;
5. Otherwise, recursively find the $(k - |A|)$-th smallest one among B;

This algorithm can be formalized in below equation. Suppose $0 < k \leq |L|$, where L is a non-empty list of elements. Denote l_1 as the first element in L. It is chosen as the pivot; L' contains the rest elements except for l_1. $(A, B) = partition(\lambda_x \cdot x \leq l_1, L')$. It partitions L' by using the same algorithm defined in the chapter of quick sort.

$$top(k, L) = \begin{cases} l_1 & : & |A| = k - 1 \\ top(k - 1 - |A|, B) & : & |A| < k - 1 \\ top(k, A) & : & otherwise \end{cases} \tag{14.3}$$

$$partition(p, L) = \begin{cases} (\Phi, \Phi) & : & L = \Phi \\ (\{l_1\} \cup A, B) & : & p(l_1), (A, B) = partition(p, L') \\ (A, \{l_1\} \cup B) & : & \neg p(l_1) \end{cases} \tag{14.4}$$

The following Haskell example program implements this algorithm.

```
top n (x:xs) | len == n - 1 = x
             | len < n - 1 = top (n - len - 1) bs
             | otherwise = top n as
    where
      (as, bs) = partition (≤x) xs
      len = length as
```

The partition function is provided in Haskell standard library, the detailed implementation can be referred to previous chapter about quick sort.

The lucky case is that, the k-th smallest element is selected as the pivot at the very beginning. The partition function examines the whole list, and finds that there are $k - 1$ elements not greater than the pivot, we are done in just $O(n)$ time. The worst case is that either the maximum or the minimum element is selected as the pivot every time. The partition always produces an empty sub list, that either A or B is empty. If we always pick the minimum as the pivot, the performance is bound to $O(kn)$. If we always pick the maximum as the pivot, the performance is $O((n - k)n)$. If k is much less than n, it downgrades to quadratic $O(n^2)$ time.

The best case (not the lucky case), is that the pivot always partition the list perfectly. The length of A is nearly as same as the length of B. The list is halved every time. It needs about $O(\lg n)$ partitions, each partition takes linear time proportion to the length of the halved list. This can be expressed as $O(n + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{2^m})$, where m is the smallest number satisfies $\frac{n}{2^m} < k$. Summing the series leads to the result of $O(n)$.

The average case analysis needs tool of mathematical expectation. It's quite similar to the proof given in previous chapter of quick sort. It's left as an exercise to the reader.

Similar as quick sort, this divide and conquer selection algorithm performs well most time in practice. We can take the same engineering practice such as media-of-three, or randomly select the pivot as we did for quick sort. Below is the imperative realization for example.

```
1: function TOP(k, A, l, u)
2:     EXCHANGE A[l] ↔ A[ RANDOM(l, u) ]          ▷ Randomly select in [l, u]
3:     p ← PARTITION(A, l, u)
4:     if p − l + 1 = k then
5:         return A[p]
6:     if k < p − l + 1 then
7:         return TOP(k, A, l, p − 1)
8:     return TOP(k − p + l − 1, A, p + 1, u)
```

This algorithm searches the k-th smallest element in range of $[l, u]$ for array A. The boundaries are included. It first randomly selects a position, and swaps it with the first one. Then this element is chosen as the pivot for partitioning. The partition algorithm in-place moves elements and returns the position where the pivot being moved. If the pivot is just located at position k, then we are done; if there are more than $k - 1$ elements not greater than the pivot, the algorithm recursively searches the k-th smallest one in range $[l, p-1]$; otherwise, k is deduced by the number of elements before the pivot, and recursively searches the range after the pivot $[p + 1, u]$.

There are many methods to realize the partition algorithm, below one is based on N. Lumoto's method. Other realizations are left as exercises to the reader.

```
1: function PARTITION(A, l, u)
2:     p ← A[l]
3:     L ← l
4:     for R ← l + 1 to u do
5:         if ¬(p < A[R]) then
6:             L ← L + 1
7:             EXCHANGE A[L] ↔ A[R]
8:     EXCHANGE A[L] ↔ p
9:     return L
```

Below ANSI C example program implements this algorithm. Note that it handles the special case that either the array is empty, or k is out of the boundaries of the array. It returns -1 to indicate the search failure.

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

/* The result is stored in xs[k], returns k if u-l ≥k, otherwise -1 */
```

```
int top(int k, Key* xs, int l, int u) {
    int p;
    if (l < u) {
        swap(xs, l, rand() % (u - l) + l);
        p = partition(xs, l, u);
        if (p - l + 1 == k)
            return p;
        return (k < p - l + 1) ? top(k, xs, l, p) :
                                 top(k- p + l - 1, xs, p + 1, u);
    }
    return -1;
}
```

There is a method proposed by Blum, Floyd, Pratt, Rivest and Tarjan in 1973, which ensures the worst case performance being bound to $O(n)$ [2], [3]. It divides the list into small groups. Each group contains no more than 5 elements. The median of each group among these 5 elements are identified quickly. Then there are $\frac{n}{5}$ median elements selected. We repeat this step, and divide them again into groups of 5, and recursively select the *median of median*. It's obviously that the final 'true' median can be found in $O(\lg n)$ time. This is the best pivot for partitioning the list. Next, we halve the list by this pivot and recursively search for the k-th smallest one. The performance can be calculated as the following.

$$T(n) = c_1 lgn + c_2 n + T(\frac{n}{2}) \tag{14.5}$$

Where c_1 and c_2 are constant factors for the median of median and partition computation respectively. Solving this equation with telescope method or the master theory in [2] gives the linear $O(n)$ performance. The detailed algorithm realization is left as exercise to the reader.

In case we just want to pick the top k smallest elements, but don't care about the order of them, the algorithm can be adjusted a little bit to fit.

$$tops(k, L) = \begin{cases} \Phi & : & k = 0 \vee L = \Phi \\ A & : & |A| = k \\ A \cup \{l_1\} \cup tops(k - |A| - 1, B) & : & |A| < k \\ tops(k, A) & : & otherwise \end{cases} \tag{14.6}$$

Where A, B have the same meaning as before that, $(A, B) = partition(\lambda_x \cdot x \leq l_1, L')$ if L isn't empty. The relative example program in Haskell is given as below.

```
tops _ [] = []
tops 0 _  = []
tops n (x:xs) | len == n = as
              | len < n  = as ++ [x] ++ tops (n-len-1) bs
              | otherwise = tops n as
    where
      (as, bs) = partition (≤ x) xs
      len = length as
```

binary search

Another popular divide and conquer algorithm is binary search. We've shown it in the chapter about insertion sort. When I was in school, the teacher who taught math played a magic to me, He asked me to consider a natural number less than 1000. Then he asked me some questions, I only replied 'yes' or 'no', and finally he guessed my number. He typically asked questions like the following:

- Is it an even number?
- Is it a prime number?
- Are all digits same?
- Can it be divided by 3?
- ...

Most of the time he guessed the number within 10 questions. My classmates and I all thought it's unbelievable.

This game will not be so interesting if it downgrades to a popular TV program, that the price of a product is hidden, and you must figure out the exact price in 30 seconds. The host of the program tells you if your guess is higher or lower to the fact. If you win, the product is yours. The best strategy is to use similar divide and conquer approach to perform a binary search. So it's common to find such conversation between the player and the host:

- P: 1000;
- H: High;
- P: 500;
- H: Low;
- P: 750;
- H: Low;
- P: 890;
- H: Low;
- P: 990;
- H: Bingo.

My math teacher told us that, because the number we considered is within 1000, if he can halve the numbers every time by designing good questions, the number will be found in 10 questions. This is because $2^{10} = 1024 > 1000$. However, it would be boring to just ask it is higher than 500, is lower than 250, ... Actually, the question 'is it even' is very good, because it always halve the numbers.

Come back to the binary search algorithm. It is only applicable to a sequence of ordered number. I've seen programmers tried to apply it to unsorted array, and took several hours to figure out why it doesn't work. The idea is quite

straightforward, in order to find a number x in an ordered sequence A, we firstly check middle point number, compare it with x, if they are same, then we are done; If x is smaller, as A is ordered, we need only recursively search it among the first half; otherwise we search it among the second half. Once A gets empty and we haven't found x yet, it means x doesn't exist.

Before formalizing this algorithm, there is a surprising fact need to be noted. Donald Knuth stated that 'Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky'. Jon Bentley pointed out that most binary search implementation contains errors, and even the one given by him in the first version of 'Programming pearls' contains an error undetected over twenty years [4].

There are two kinds of realization, one is recursive, the other is iterative. The recursive solution is as same as what we described. Suppose the lower and upper boundaries of the array are l and u inclusive.

```
1: function BINARY-SEARCH(x, A, l, u)
2:     if u < l then
3:         Not found error
4:     else
5:         m ← l + ⌊(u−l)/2⌋                    ▷ avoid overflow of ⌊(l+u)/2⌋
6:         if A[m] = x then
7:             return m
8:         if x < A[m] then
9:             return BINARY-SEARCH(x, A, l, m - 1)
10:        else
11:            return BINARY-SEARCH(x, A, m + 1, u)
```

As the comment highlights, if the integer is represented with limited words, we can't merely use $\lfloor \frac{l+u}{2} \rfloor$ because it may cause overflow if l and u are big.

Binary search can also be realized in iterative manner, that we keep updating the boundaries according to the middle point comparison result.

```
1: function BINARY-SEARCH(x, A, l, u)
2:     while l < u do
3:         m ← l + ⌊(u−l)/2⌋
4:         if A[m] = x then
5:             return m
6:         if x < A[m] then
7:             u ← m − 1
8:         else
9:             l ← m + 1
       return NIL
```

The implementation is very good exercise, we left it to the reader. Please try all kinds of methods to verify your program.

Since the array is halved every time, the performance of binary search is bound to $O(\lg n)$ time.

In purely functional settings, the list is represented with singly linked-list. It's linear time to randomly access the element for a given position. Binary search doesn't make sense in such case. However, it good to analyze what the performance will downgrade to. Consider the following equation.

$$bsearch(x, L) = \begin{cases} Err & : & L = \Phi \\ b_1 & : & x = b_1, (A, B) = splitAt(\lfloor \frac{|L|}{2} \rfloor, L) \\ bsearch(x, A) & : & B = \Phi \vee x < b_1 \\ bsearch(x, B') & : & otherwise \end{cases}$$

Where b_1 is the first element if B isn't empty, and B' holds the rest except for b_1. The *splitAt* function takes $O(n)$ time to divide the list into two subs A and B (see the appendix A, and the chapter about merge sort for detail). If B isn't empty and x is equal to b_1, the search returns; Otherwise if it is less than b_1, as the list is sorted, we need recursively search in A, otherwise, we search in B. If the list is empty, we raise error to indicate search failure.

As we always split the list in the middle point, the number of elements halves in each recursion. In every recursive call, we takes linear time for splitting. The splitting function only traverses the first half of the linked-list, Thus the total time can be expressed as.

$$T(n) = c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} + ...$$

This results $O(n)$ time, which is as same as the brute force search from head to tail:

$$search(x, L) = \begin{cases} Err & : & L = \Phi \\ l_1 & : & x = l_1 \\ search(x, L') & : & otherwise \end{cases}$$

As we mentioned in the chapter about insertion sort, the functional approach of binary search is through binary search tree. That the ordered sequence is represented in a tree (self balanced tree if necessary), which offers logarithm time searching [2].

Although it doesn't make sense to apply divide and conquer binary sort on linked-list, binary search can still be very useful in purely functional settings. Consider solving an equation $a^x = y$, for given natural numbers a and y, where $a \leq y$. We want to find the integer solution for x if there is. Of course brute-force naive searching can solve it. We can examine all numbers one by one from 0 for $a^0, a^1, a^2, ...$, stops if $a^i = y$ or report that there is no solution if $a^i < y < a^{i+1}$ for some i. We initialize the solution domain as $X = \{0, 1, 2, ...\}$, and call the below exhausted searching function $solve(a, y, X)$.

$$solve(a, y, X) = \begin{cases} x_1 & : & a^{x_1} = y \\ solve(a, y, X') & : & a^{x_1} < y \\ Err & : & otherwise \end{cases}$$

This function examines the solution domain in monotonic increasing order. It takes the first candidate element x_1 from X, compare a^{x_1} and y, if they are equal, then x_1 is the solution and we are done; if it is less than y, then x_1 is dropped, and we search among the rest elements represented as X'; Otherwise, since $f(x) = a^x$ is non-decreasing function when a is natural number, so the rest

[2]Some readers may argue that array should be used instead of linked-list, for example in Haskell. This book only deals with purely functional sequences in finger-tree. Different from the Haskell array, it can't support constant time random accessing

elements will only make $f(x)$ bigger and bigger. There is no integer solution for this equation. The function returns error to indicate no solution.

The computation of a^x is expensive for big a and x if precession must be kept[3]. Can it be improved so that we can compute as less as possible? The divide and conquer binary search can help. Actually, we can estimate the upper limit of the solution domain. As $a^y \leq y$, We can search in range $\{0, 1, ..., y\}$. As the function $f(x) = a^x$ is non-decreasing against its argument x, we can firstly check the middle point candidate $x_m = \lfloor \frac{0+y}{2} \rfloor$, if $a^{x_m} = y$, the solution is found; if it is less than y, we can drop all candidate solutions before x_m; otherwise we drop all candidate solutions after it; Both halve the solution domain. We repeat this approach until either the solution is found or the solution domain becomes empty, which indicates there is no integer solution.

The binary search method can be formalized as the following equation. The non-decreasing function is abstracted as a parameter. To solve our problem, we can just call it as $bsearch(f, y, 0, y)$, where $f(x) = a^x$.

$$bsearch(f, y, l, u) = \begin{cases} Err & : & u < l \\ m & : & f(m) = y, m = \lfloor \frac{l+u}{2} \rfloor \\ bsearch(f, y, l, m-1) & : & f(m) > y \\ bsearch(f, y, m+1, u) & : & f(m) < y \end{cases} \tag{14.7}$$

As we halve the solution domain in every recursion, this method computes $f(x)$ in $O(\log y)$ times. It is much faster than the brute-force searching.

2 dimensions search

It's quite natural to think that the idea of binary search can be extended to 2 dimensions or even more general – multiple-dimensions domain. However, it is not so easy.

Consider the example of a $m \times n$ matrix M. The elements in each row and each column are in strict increasing order. Figure 14.1 illustrates such a matrix for example.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & ... \\ 2 & 4 & 5 & 6 & ... \\ 3 & 5 & 7 & 8 & ... \\ 4 & 6 & 8 & 9 & ..... \end{bmatrix}$$

Figure 14.1: A matrix in strict increasing order for each row and column.

Given a value x, how to locate all elements equal to x in the matrix quickly? We need develop an algorithm, which returns a list of locations (i, j) so that $M_{i,j} = x$.

Richard Bird in [1] mentioned that he used this problem to interview candidates for entry to Oxford. The interesting story was that, those who had some

[3]One alternative is to reuse the result of a^n when compute $a^{n+1} = aa^n$. Here we consider for general form monotonic function $f(n)$

computer background at school tended to use binary search. But it's easy to get stuck.

The usual way follows binary search idea is to examine element at $M_{\frac{m}{2},\frac{n}{2}}$. If it is less than x, we can only drop the elements in the top-left area; If it is greater than x, only the bottom-right area can be dropped. Both cases are illustrated in figure 14.2, the gray areas indicate elements can be dropped.

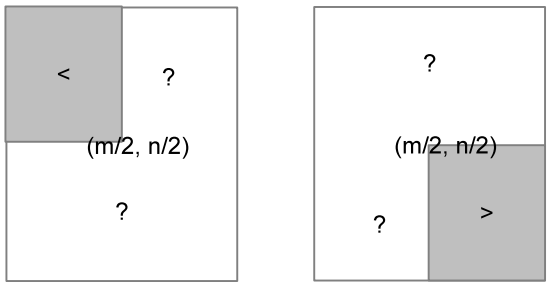


Figure 14.2: Left: the middle point element is smaller than x. All elements in the gray area are less than x; Right: the middle point element is greater than x. All elements in the gray area are greater than x.

The problem is that the solution domain changes from a rectangle to a 'L' shape in both cases. We can't just recursively apply search on it. In order to solve this problem systematically, we define the problem more generally, using brute-force search as a start point, and keep improving it bit by bit.

Consider a function $f(x, y)$, which is strict increasing for its arguments, for instance $f(x, y) = a^x + b^y$, where a and b are natural numbers. Given a value z, which is a natural number too, we want to solve the equation $f(x, y) = z$ by finding all candidate pairs (x, y).

With this definition, the matrix search problem can be specialized by below function.

$$f(x, y) = \begin{cases} M_{x,y} & : \quad 1 \leq x \leq m, 1 \leq y \leq n \\ -1 & : \quad otherwise \end{cases}$$

Brute-force 2D search

As all solutions should be found for $f(x, y)$. One can immediately give the brute force solution by embedded looping.

```
1: function SOLVE(f, z)
2:     A ← Φ
3:     for x ∈ {0, 1, 2, ..., z} do
4:         for y ∈ {0, 1, 2, ..., z} do
5:             if f(x, y) = z then
6:                 A ← A ∪ {(x, y)}
7:     return A
```

This definitely calculates f for $(z+1)^2$ times. It can be formalized as in (14.8).

$$solve(f, z) = \{(x, y) | x \in \{0, 1, ..., z\}, y \in \{0, 1, ..., z\}, f(x, y) = z\} \tag{14.8}$$

Saddleback search

We haven't utilize the fact that $f(x, y)$ is strict increasing yet. Dijkstra pointed out in [6], instead of searching from bottom-left corner, starting from the top-left leads to one effective solution. As illustrated in figure 14.3, the search starts from $(0, z)$, for every point (p, q), we compare $f(p, q)$ with z:

- If $f(p, q) < z$, since f is strict increasing, for all $0 \leq y < q$, we have $f(p, y) < z$. We can drop all points in the vertical line section (in red color);
- If $f(p, q) > z$, then $f(x, q) > z$ for all $p < x \leq z$. We can drop all points in the horizontal line section (in blue color);
- Otherwise if $f(p, q) = z$, we mark (p, q) as one solution, then both line sections can be dropped.

This is a systematical way to scale down the solution domain rectangle. We keep dropping a row, or a column, or both.

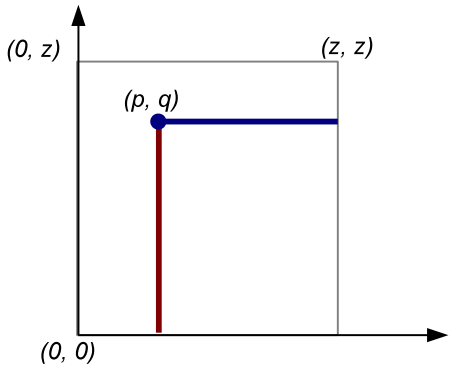


Figure 14.3: Search from top-left.

This method can be formalized as a function $search(f, z, p, q)$, which searches solutions for equation $f(x, y) = z$ in rectangle with top-left corner (p, q), and bottom-right corner $(z, 0)$. We start the searching by initializing $(p, q) = (0, z)$ as $solve(f, z) = search(f, z, 0, z)$

$$search(f, z, p, q) = \begin{cases} \Phi & : & p > z \vee q < 0 \\ search(f, z, p + 1, q) & : & f(p, q) < z \\ search(f, z, p, q - 1) & : & f(p, q) > z \\ \{(p, q)\} \cup search(f, z, p + 1, q - 1) & : & otherwise \end{cases} \tag{14.9}$$

The first clause is the edge case, there is no solution if (p, q) isn't top-left to $(z, 0)$. The following example Haskell program implements this algorithm.

```
solve f z = search 0 z where
  search p q | p > z || q < 0 = []
             | z' < z = search (p + 1) q
             | z' > z = search p (q - 1)
             | otherwise = (p, q) : search (p + 1) (q - 1)
    where z' = f p q
```

Considering the calculation of f may be expensive, this program stores the result of $f(p, q)$ to variable z'. This algorithm can also be implemented in iterative manner, that the boundaries of solution domain keeps being updated in a loop.

1: **function** SOLVE(f, z)
2: $p \leftarrow 0, q \leftarrow z$
3: $S \leftarrow \Phi$
4: **while** $p \leq z \wedge q \geq 0$ **do**
5: $z' \leftarrow f(p, q)$
6: **if** $z' < z$ **then**
7: $p \leftarrow p + 1$
8: **else if** $z' > z$ **then**
9: $q \leftarrow q - 1$
10: **else**
11: $S \leftarrow S \cup \{(p, q)\}$
12: $p \leftarrow p + 1, q \leftarrow q - 1$
13: **return** S

It's intuitive to translate this imperative algorithm to real program, as the following example Python code.

```
def solve(f, z):
    (p, q) = (0, z)
    res = []
    while p ≤ z and q ≥ 0:
        z1 = f(p, q)
        if z1 < z:
            p = p + 1
        elif z1 > z:
            q = q - 1
        else:
            res.append((p, q))
            (p, q) = (p + 1, q - 1)
    return res
```

It is clear that in every iteration, At least one of p and q advances to the bottom-right corner by one. Thus it takes at most $2(z + 1)$ steps to complete searching. This is the worst case. There are three best cases. The first one happens that in every iteration, both p and q advance by one, so that it needs only $z+1$ steps; The second case keeps advancing horizontally to right and ends when p exceeds z; The last case is similar, that it keeps moving down vertically to the bottom until q becomes negative.

Figure 14.4 illustrates the best cases and the worst cases respectively. Figure 14.4 (a) is the case that every point $(x, z-x)$ in diagonal satisfies $f(x, z-x) = z$,

it uses $z+1$ steps to arrive at $(z, 0)$; (b) is the case that every point (x, z) along the top horizontal line gives the result $f(x, z) < z$, the algorithm takes $z + 1$ steps to finish; (c) is the case that every point $(0, x)$ along the left vertical line gives the result $f(0, x) > z$, thus the algorithm takes $z + 1$ steps to finish; (d) is the worst case. If we project all the horizontal sections along the search path to x axis, and all the vertical sections to y axis, it gives the total steps of $2(z + 1)$.

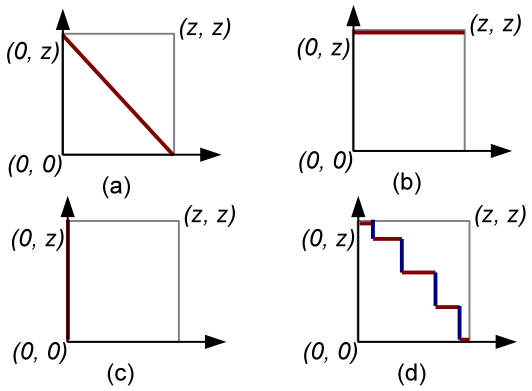


Figure 14.4: The best cases and the worst cases.

Compare to the quadratic brute-force method ($O(z^2)$), we improve to a linear algorithm bound to $O(z)$.

Bird imagined that the name 'saddleback' is because the 3D plot of f with the smallest bottom-left and the latest top-right and two wings looks like a saddle as shown in figure 14.5

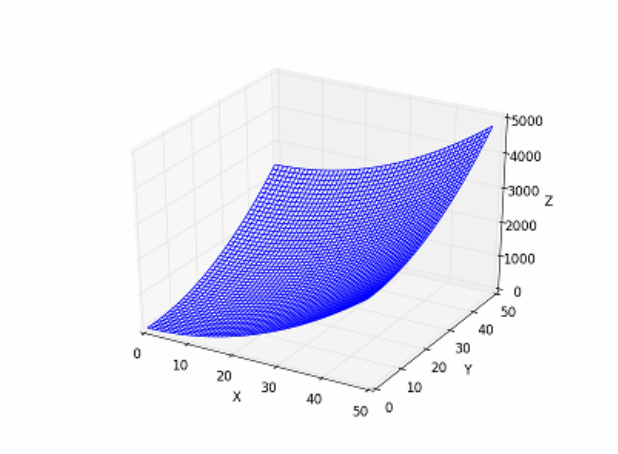


Figure 14.5: Plot of $f(x, y) = x^2 + y^2$.

Improved saddleback search

We haven't utilized the binary search tool so far, even the problem extends to 2-dimension domain. The basic saddleback search starts from the top-left corner

$(0, z)$ to the bottom-right corner $(z, 0)$. This is actually over-general domain. we can constraint it a bit more accurate.

Since f is strict increasing, we can find the biggest number m, that $0 \leq m \leq z$, along the y axis which satisfies $f(0, m) \leq z$; Similarly, we can find the biggest n, that $0 \leq n \leq z$, along the x axis, which satisfies $f(n, 0) \leq z$; And the solution domain shrinks from $(0, z) - (z, 0)$ to $(0, m) - (n, 0)$ as shown in figure 14.6.

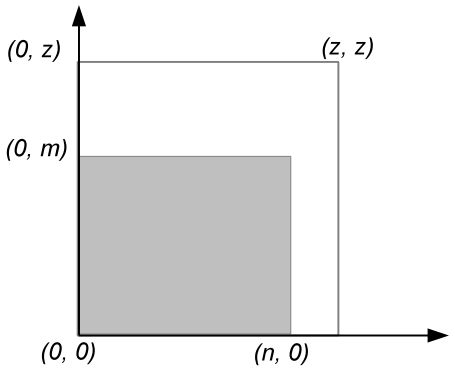


Figure 14.6: A more accurate search domain shown in gray color.

Of course m and n can be found by brute-force like below.

$$\begin{aligned} m &= max(\{y | 0 \leq y \leq z, f(0, y) \leq z\}) \\ n &= max(\{x | 0 \leq x \leq z, f(x, 0) \leq z\}) \end{aligned} \tag{14.10}$$

When searching m, the x variable of f is bound to 0. It turns to be one dimension search problem for a strict increasing function (or in functional term, a Curried function $f(0, y)$). Binary search works in such case. However, we need a bit modification for equation (14.7). Different from searching a solution $l \leq x \leq u$, so that $f(x) = y$ for a given y; we need search for a solution $l \leq x \leq u$ so that $f(x) \leq y < f(x + 1)$.

$$bsearch(f, y, l, u) = \begin{cases} l & : & u \leq l \\ m & : & f(m) \leq y < f(m+1), m = \lfloor \frac{l+u}{2} \rfloor \\ bsearch(f, y, m+1, u) & : & f(m) \leq y \\ bsearch(f, y, l, m-1) & : & otherwise \end{cases} \tag{14.11}$$

The first clause handles the edge case of empty range. The lower boundary is returned in such case; If the middle point produces a value less than or equal to the target, while the next one evaluates to a bigger value, then the middle point is what we are looking for; Otherwise if the point next to the middle also evaluates to a value not greater than the target, the lower bound is increased by one, and we perform recursively binary search; In the last case, the middle point evaluates to a value greater than the target, upper bound is updated as the point proceeds to the middle for further recursive searching. The following Haskell example code implements this modified binary search.

```
bsearch f y (l, u) | u ≤ l = l
```

```
                | f m ≤ y = if f (m + 1) ≤ y
                            then bsearch f y (m + 1, u) else m
                | otherwise = bsearch f y (l, m-1)
  where m = (l + u) `div` 2
```

Then m and n can be found with this binary search function.

$$\begin{aligned} m &= bsearch(\lambda_y \cdot f(0, y), z, 0, z) \\ n &= bsearch(\lambda_x \cdot f(x, 0), z, 0, z) \end{aligned} \tag{14.12}$$

And the improved saddleback search shrinks to this new search domain $solve(f, z) = search(f, z, 0, m)$:

$$search(f, z, p, q) = \begin{cases} \Phi & : \quad p > n \vee q < 0 \\ search(f, z, p + 1, q) & : \quad f(p, q) < z \\ search(f, z, p, q - 1) & : \quad f(p, q) > z \\ \{(p, q)\} \cup search(f, z, p + 1, q - 1) & : \quad otherwise \end{cases} \tag{14.13}$$

It's almost as same as the basic saddleback version, except that it stops if p exceeds n, but not z. In real implementation, the result of $f(p, q)$ can be calculated once, and stored in a variable as shown in the following Haskell example.

```
solve' f z = search 0 m where
  search p q | p > n || q < 0 = []
             | z' < z = search (p + 1) q
             | z' > z = search p (q - 1)
             | otherwise = (p, q) : search (p + 1) (q - 1)
    where z' = f p q
  m = bsearch (f 0) z (0, z)
  n = bsearch (λx→f x 0) z (0, z)
```

This improved saddleback search firstly performs binary search two rounds to find the proper m, and n. Each round is bound to $O(\lg z)$ times of calculation for f; After that, it takes $O(m + n)$ time in the worst case; and $O(min(m, n))$ time in the best case. The overall performance is given in the following table.

	times of evaluation f
worst case	$2 \log z + m + n$
best case	$2 \log z + min(m, n)$

For some function $f(x, y) = a^x + b^y$, for positive integers a and b, m and n will be relative small, that the performance is close to $O(\lg z)$.

This algorithm can also be realized in imperative approach. Firstly, the binary search should be modified.

```
1: function BINARY-SEARCH(f, y, (l, u))
2:     while l < u do
3:         m ← ⌊(l+u)/2⌋
4:         if f(m) ≤ y then
5:             if y < f(m + 1) then
6:                 return m
7:             l ← m + 1
8:         else
9:             u ← m
```

```
10:     return l
```

Utilize this algorithm, the boundaries m and n can be found before performing the saddleback search.

```
1: function SOLVE(f, z)
2:     m ← BINARY-SEARCH(λ_y · f(0, y), z, (0, z))
3:     n ← BINARY-SEARCH(λ_x · f(x, 0), z, (0, z))
4:     p ← 0, q ← m
5:     S ← Φ
6:     while p ≤ n ∧ q ≥ 0 do
7:         z' ← f(p, q)
8:         if z' < z then
9:             p ← p + 1
10:        else if z' > z then
11:            q ← q − 1
12:        else
13:            S ← S ∪ {(p, q)}
14:            p ← p + 1, q ← q − 1
15:    return S
```

The implementation is left as exercise to the reader.

More improvement to saddleback search

In figure 14.2, two cases are shown for comparing the value of the middle point in a matrix with the given value. One case is the center value is smaller than the given value, the other is bigger. In both cases, we can only throw away $\frac{1}{4}$ candidates, and left a 'L' shape for further searching.

Actually, one important case is missing. We can extend the observation to any point inside the rectangle searching area. As shown in the figure 14.7.

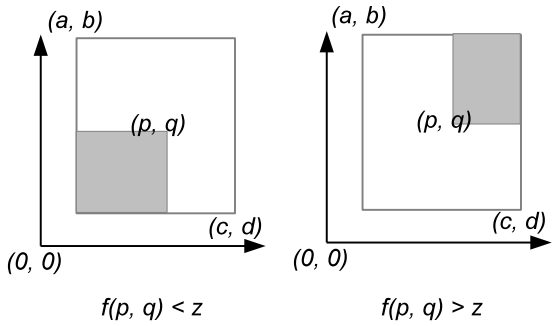
Suppose we are searching in a rectangle from the lower-left corner (a, b) to the upper-right corner (c, d). If the (p, q) isn't the middle point, and $f(p, q) \neq z$. We can't ensure the area to be dropped is always $1/4$. However, if $f(p, q) = z$, as f is strict increasing, we are not only sure both the lower-left and the upper-right sub areas can be thrown, but also all the other points in the column p and row q. The problem can be scaled down fast, because only $1/2$ area is left.

This indicates us, instead of jumping to the middle point to start searching. A more efficient way is to find a point which evaluates to the target value. One straightforward way to find such a point, is to perform binary search along the center horizontal line or the center vertical line of the rectangle.

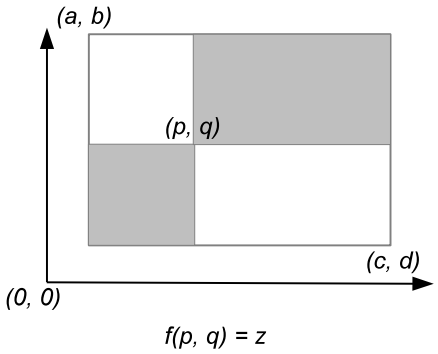
The performance of binary search along a line is logarithmic to the length of that line. A good idea is to always pick the shorter center line as shown in figure 14.8. That if the height of the rectangle is longer than the width, we perform binary search along the horizontal center line; otherwise we choose the vertical center line.

However, what if we can't find a point (p, q) in the center line, that satisfies $f(p, q) = z$? Let's take the center horizontal line for example. even in such case, we can still find a point that $f(p, q) < z < f(p + 1, q)$. The only difference is that we can't drop the points in row p and q completely.

Combine this conditions, the binary search along the horizontally line is to find a p, satisfies $f(p, q) \leq z < f(p + 1, q)$; While the vertical line search



(a) If $f(p, q) \neq z$, only lower-left or upper-right sub area (in gray color) can be thrown. Both left a 'L' shape.



(b) If $f(p, q) = z$, both sub areas can be thrown, the scale of the problem is halved.

Figure 14.7: The efficiency of scaling down the search domain.

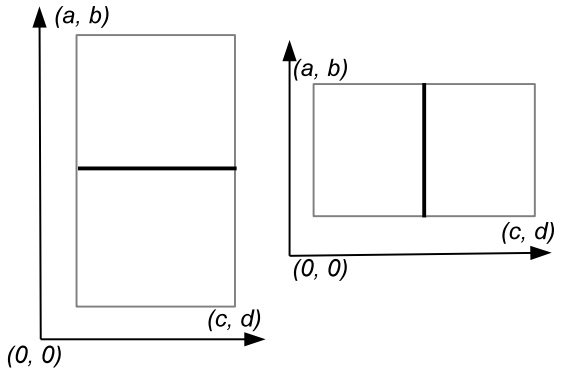


Figure 14.8: Binary search along the shorter center line.

condition is $f(p, q) \leq z < f(p, q + 1)$.

The modified binary search ensures that, if all points in the line segment give $f(p, q) < z$, the upper bound will be found; and the lower bound will be found if they all greater than z. We can drop the whole area on one side of the center line in such case.

Sum up all the ideas, we can develop the efficient improved saddleback search as the following.

1. Perform binary search along the y axis and x axis to find the tight boundaries from $(0, m)$ to $(n, 0)$;
2. Denote the candidate rectangle as $(a, b) - (c, d)$, if the candidate rectangle is empty, the solution is empty;
3. If the height of the rectangle is longer than the width, perform binary search along the center horizontal line; otherwise, perform binary search along the center vertical line; denote the search result as (p, q);
4. If $f(p, q) = z$, record (p, q) as a solution, and recursively search two sub rectangles $(a, b) - (p - 1, q + 1)$ and $(p + 1, q - 1) - (c, d)$;
5. Otherwise, $f(p, q) \neq z$, recursively search the same two sub rectangles plus a line section. The line section is either $(p, q + 1) - (p, b)$ as shown in figure 14.9 (a); or $(p + 1, q) - (c, q)$ as shown in figure 14.9 (b).

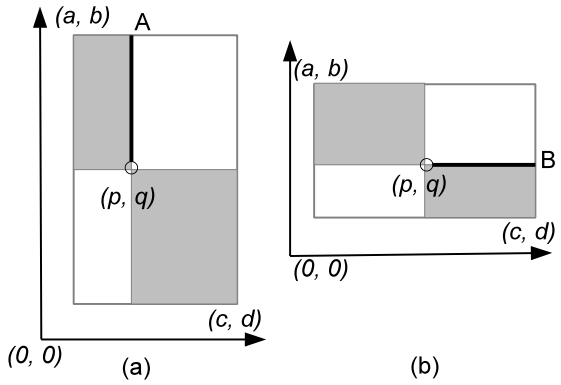


Figure 14.9: Recursively search the gray areas, the bold line should be included if $f(p, q) \neq z$.

This algorithm can be formalized as the following. The equation (14.11), and (14.12) are as same as before. A new *search* function should be defined.

Define $Search_{(a,b),(c,d)}$ as a function for searching rectangle with top-left corner (a, b), and bottom-right corner (c, d).

$$search_{(a,b),(c,d)} = \begin{cases} \Phi & : & c < a \vee d < b \\ csearch & : & c - a < b - d \\ rsearch & : & otherwise \end{cases} \tag{14.14}$$

Function *csearch* performs binary search in the center horizontal line to find a point (p, q) that $f(p, q) \leq z < f(p + 1, q)$. This is shown in figure 14.9

(a). There is a special edge case, that all points in the lines evaluate to values greater than z. The general binary search will return the lower bound as result, so that $(p, q) = (a, \lfloor \frac{b+d}{2} \rfloor)$. The whole upper side includes the center line can be dropped as shown in figure 14.10 (a).

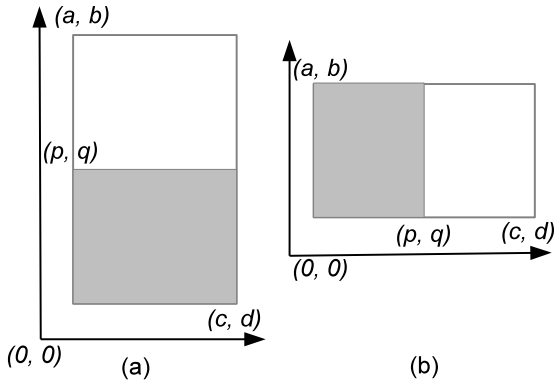


Figure 14.10: Edge cases when performing binary search in the center line.

$$csearch = \begin{cases} search_{(p,q-1),(c,d)} & : & z < f(p,q) \\ search_{(a,b),(p-1,q+1)} \cup \{(p,q)\} \cup search_{(p+1,q-1),(c,d)} & : & f(p,q) = z \\ search_{(a,b),(p,q+1)} \cup search_{(p+1,q-1),(c,d)} & : & otherwise \end{cases} \tag{14.15}$$

Where

$$\begin{aligned} q &= \lfloor \tfrac{b+d}{2} \rfloor) \\ p &= bsearch(\lambda_x \cdot f(x, q), z, (a, c)) \end{aligned}$$

Function $rsearch$ is quite similar except that it searches in the center horizontal line.

$$rsearch = \begin{cases} search_{(a,b),(p-1,q)} & : & z < f(p,q) \\ search_{(a,b),(p-1,q+1)} \cup \{(p,q)\} \cup search_{(p+1,q-1),(c,d)} & : & f(p,q) = z \\ search_{(a,b),(p-1,q+1)} \cup search_{(p+1,q),(c,d)} & : & otherwise \end{cases} \tag{14.16}$$

Where

$$\begin{aligned} p &= \lfloor \tfrac{a+c}{2} \rfloor) \\ q &= bsearch(\lambda_y \cdot f(p, y), z, (d, b)) \end{aligned}$$

The following Haskell program implements this algorithm.

```
search f z (a, b) (c, d) | c < a || b < d = []
                         | c - a < b - d = let q = (b + d) `div` 2 in
                                           csearch (bsearch (λx → f x q) z (a, c), q)
                         | otherwise = let p = (a + c) `div` 2 in
                                       rsearch (p, bsearch (f p) z (d, b))
  where
    csearch (p, q) | z < f p q = search f z (p, q - 1) (c, d)
```

```
                | f p q == z = search f z (a, b) (p - 1, q + 1) ++
                                 (p, q) : search f z (p + 1, q - 1) (c, d)
                | otherwise = search f z (a, b) (p, q + 1) ++
                                search f z (p + 1, q - 1) (c, d)
    rsearch (p, q) | z < f p q = search f z (a, b) (p - 1, q)
                | f p q == z = search f z (a, b) (p - 1, q + 1) ++
                                 (p, q) : search f z (p + 1, q - 1) (c, d)
                | otherwise = search f z (a, b) (p - 1, q + 1) ++
                                 search f z (p + 1, q) (c, d)
```

And the main program calls this function after performing binary search in X and Y axes.

```
solve f z = search f z (0, m) (n, 0) where
  m = bsearch (f 0) z (0, z)
  n = bsearch (λx → f x 0) z (0, z)
```

Since we drop half areas in every recursion, it takes $O(\log(mn))$ rounds of search. However, in order to locate the point (p, q), which halves the problem, we must perform binary search along the center line. which will call f about $O(\log(min(m, n)))$ times. Denote the time of searching a $m \times n$ rectangle as $T(m, n)$, the recursion relationship can be represented as the following.

$$T(m, n) = \log(min(m, n)) + 2T(\frac{m}{2}, \frac{n}{2}) \tag{14.17}$$

Suppose $m > n$, using telescope method, for $m = 2^i$, and $n = 2^j$. We have:

$$\begin{aligned} T(2^i, 2^j) &= j + 2T(2^{i-1}, 2^{j-1}) \\ &= \sum_{k=0}^{i-1} 2^k (j - k) \\ &= O(2^i(j - i)) \\ &= O(m \log(n/m)) \end{aligned} \tag{14.18}$$

Richard Bird proved that this is asymptotically optimal by a lower bound of searching a given value in $m \times n$ rectangle [1].

The imperative algorithm is almost as same as the functional version. We skip it for the sake of brevity.

Exercise 14.1

- Prove that the average case for the divide and conquer solution to k-selection problem is $O(n)$. Please refer to previous chapter about quick sort.
- Implement the imperative k-selection problem with 2-way partition, and median-of-three pivot selection.
- Implement the imperative k-selection problem to handle duplicated elements effectively.
- Realize the median-of-median k-selection algorithm and implement it in your favorite programming language.

- The $tops(k, L)$ algorithm uses list concatenation likes $A \cup \{l_1\} \cup tops(k - |A| - 1, B)$. It is linear operation which is proportion to the length of the list to be concatenated. Modify the algorithm so that the sub lists are concatenated by one pass.

- The author considered another divide and conquer solution for the k-selection problem. It finds the maximum of the first k elements and the minimum of the rest. Denote them as x, and y. If x is smaller than y, it means that all the first k elements are smaller than the rest, so that they are exactly the top k smallest; Otherwise, There are some elements in the first k should be swapped.

```
1: procedure TOPS(k, A)
2:     l ← 1
3:     u ← |A|
4:     loop
5:         i ← MAX-AT(A[l..k])
6:         j ← MIN-AT(A[k + 1..u])
7:         if A[i] < A[j] then
8:             break
9:         EXCHANGE A[l] ↔ A[j]
10:        EXCHANGE A[k + 1] ↔ A[i]
11:        l ← PARTITION(A, l, k)
12:        u ← PARTITION(A, k + 1, u)
```

  Explain why this algorithm works? What's the performance of it?

- Implement the binary search algorithm in both recursive and iterative manner, and try to verify your version automatically. You can either generate randomized data, test your program with the binary search invariant, or compare with the built-in binary search tool in your standard library.

- Implement the improved saddleback search by firstly performing binary search to find a more accurate solution domain in your favorite imperative programming language.

- Realize the improved 2D search, by performing binary search along the shorter center line, in your favorite imperative programming language.

- Someone considers that the 2D search can be designed as the following. When search a rectangle, as the minimum value is at bottom-left, and the maximum at to-right. If the target value is less than the minimum or greater than the maximum, then there is no solution; otherwise, the rectangle is divided into 4 sub rectangles at the center point, then perform recursively searching.

```
1: procedure SEARCH(f, z, a, b, c, d)        ▷ (a, b): bottom-left (c, d):
   top-right
2:     if z ≤ f(a, b) ∨ f(c, d) ≥ z then
3:         if z = f(a, b) then
4:             record (a, b) as a solution
5:         if z = f(c, d) then
```

```
6:              record (c, d) as a solution
7:          return
8:      p ← ⌊(a+c)/2⌋
9:      q ← ⌊(b+d)/2⌋
10:     SEARCH(f, z, a, q, p, d)
11:     SEARCH(f, z, p, q, c, d)
12:     SEARCH(f, z, a, b, p, q)
13:     SEARCH(f, z, p, b, c, q)
```

What's the performance of this algorithm?

14.2.2 Information reuse

One interesting behavior that is that people learning while searching. We do not only remember lessons which cause search fails, but also learn patterns which lead to success. This is a kind of information reusing, no matter the information is positive or negative. However, It's not easy to determine what information should be kept. Too little information isn't enough to help effective searching, while keeping too much is expensive in term of spaces.

In this section, we'll first introduce two interesting problems, Boyer-Moore majority number problem and the maximum sum of sub vector problem. Both reuse information as little as possible. After that, two popular string matching algorithms, Knuth-Morris-Pratt algorithm and Boyer-Moore algorithm will be introduced.

Boyer-Moore majority number

Voting is quite critical to people. We use voting to choose the leader, make decision or reject a proposal. In the months when I was writing this chapter, there are three countries in the world voted their presidents. All of the three voting activities utilized computer to calculate the result.

Suppose there is a country in a small island wants a new president. According to the constitution, only if the candidate wins more than half of the votes can be selected as the president. Given a serious of votes, such as A, B, A, C, B, B, D, ..., can we develop a program tells who is the new president if there is, or indicate nobody wins more than half of the votes?

Of course this problem can be solved with brute-force by using a map. As what we did in the chapter of binary search tree[4].

```
template<typename T>
T majority(const T* xs, int n, T fail) {
    map<T, int> m;
    int i, max = 0;
    T r;
    for (i = 0; i < n; ++i)
        ++m[xs[i]];
    for (typename map<T, int>::iterator it = m.begin(); it != m.end(); ++it)
        if (it->second > max) {
            max = it->second;
```

[4]There is a probabilistic sub-linear space counting algorithm published in 2004, named as 'Count-min sketch'[8].

```
            r = it→first;
        }
    return max * 2 > n ? r : fail;
}
```

This program first scan the votes, and accumulates the number of votes for each individual with a map. After that, it traverse the map to find the one with the most of votes. If the number is bigger than the half, the winner is found otherwise, it returns a value to indicate fail.

The following pseudo code describes this algorithm.

```
1: function MAJORITY(A)
2:     M ← empty map
3:     for ∀a ∈ A do
4:         PUT(M, a, 1+ GET(M, a))
5:     max ← 0, m ← NIL
6:     for ∀(k, v) ∈ M do
7:         if max < v then
8:             max ← v, m ← k
9:     if max > |A|50% then
10:        return m
11:    else
12:        fail
```

For m individuals and n votes, this program firstly takes about $O(n \log m)$ time to build the map if the map is implemented in self balanced tree (red-black tree for instance); or about $O(n)$ time if the map is hash table based. However, the hash table needs more space. Next the program takes $O(m)$ time to traverse the map, and find the majority vote. The following table lists the time and space performance for different maps.

map	time	space
self-balanced tree	$O(n \log m)$	$O(m)$
hashing	$O(n)$	$O(m)$ at least

Boyer and Moore invented a cleaver algorithm in 1980, which can pick the majority element with only one scan if there is. Their algorithm only needs $O(1)$ space [7].

The idea is to record the first candidate as the winner so far, and mark him with 1 vote. During the scan process, if the winner being selected gets another vote, we just increase the vote counter; otherwise, it means somebody vote against this candidate, so the vote counter should be decreased by one. If the vote counter becomes zero, it means this candidate is voted out; We select the next candidate as the new winner and repeat the above scanning process.

Suppose there is a series of votes: A, B, C, B, B, C, A, B, A, B, B, D, B. Below table illustrates the steps of this processing.

winner	count	scan position
A	1	**A**, B, C, B, B, C, A, B, A, B, B, D, B
A	0	A, **B**, C, B, B, C, A, B, A, B, B, D, B
C	1	A, B, **C**, B, B, C, A, B, A, B, B, D, B
C	0	A, B, C, **B**, B, C, A, B, A, B, B, D, B
B	1	A, B, C, B, **B**, C, A, B, A, B, B, D, B
B	0	A, B, C, B, B, **C**, A, B, A, B, B, D, B
A	1	A, B, C, B, B, C, **A**, B, A, B, B, D, B
A	0	A, B, C, B, B, C, A, **B**, A, B, B, D, B
A	1	A, B, C, B, B, C, A, B, **A**, B, B, D, B
A	0	A, B, C, B, B, C, A, B, A, **B**, B, D, B
B	1	A, B, C, B, B, C, A, B, A, B, **B**, D, B
B	0	A, B, C, B, B, C, A, B, A, B, B, **D**, B
B	1	A, B, C, B, B, C, A, B, A, B, B, D, **B**

The key point is that, if there exits the majority greater than 50%, it can't be voted out by all the others. However, if there are not any candidates win more than half of the votes, the recorded 'winner' is invalid. Thus it is necessary to perform a second round scan for verification.

The following pseudo code illustrates this algorithm.

```
1: function MAJORITY(A)
2:     c ← 0
3:     for i ← 1 to |A| do
4:         if c = 0 then
5:             x ← A[i]
6:         if A[i] = x then
7:             c ← c + 1
8:         else
9:             c ← c − 1
10:    return x
```

If there is the majority element, this algorithm takes one pass to scan the votes. In every iteration, it either increases or decreases the counter according to the vote is support or against the current selection. If the counter becomes zero, it means the current selection is voted out. So the new one is selected as the updated candidate for further scan.

The process is linear $O(n)$ time, and the spaces needed are just two variables. One for recording the selected candidate so far, the other is for vote counting.

Although this algorithm can find the majority element if there is. it still picks an element even there isn't. The following modified algorithm verifies the final result with another round of scan.

```
1: function MAJORITY(A)
2:     c ← 0
3:     for i ← 1 to |A| do
4:         if c = 0 then
5:             x ← A[i]
6:         if A[i] = x then
7:             c ← c + 1
8:         else
9:             c ← c − 1
```

```
10:     c ← 0
11:     for i ← 1 to |A| do
12:         if A[i] = x then
13:             c ← c + 1
14:     if c > %50|A| then
15:         return x
16:     else
17:         fail
```

Even with this verification process, the algorithm is still bound to $O(n)$ time, and the space needed is constant. The following ISO C++ program implements this algorithm [5].

```
template<typename T>
T majority(const T* xs, int n, T fail) {
    T m;
    int i, c;
    for (i = 0, c = 0; i < n; ++i) {
        if (!c)
            m = xs[i];
        c += xs[i] == m ? 1 : -1;
    }
    for (i = 0, c = 0; i < n; ++i, c += xs[i] == m);
    return c * 2 > n ? m : fail;
}
```

Boyer-Moore majority algorithm can also be realized in purely functional approach. Different from the imperative settings, which use variables to record and update information, accumulators are used to define the core algorithm. Define function $maj(c, n, L)$, which takes a list of votes L, a selected candidate c so far, and a counter n. For non empty list L, we initialize c as the first vote l_1, and set the counter as 1 to start the algorithm: $maj(l_1, 1, L')$, where L' is the rest votes except for l_1. Below are the definition of this function.

$$maj(c, n, L) = \begin{cases} c & : & L = \Phi \\ maj(c, n+1, L') & : & l_1 = c \\ maj(l_1, 1, L') & : & n = 0 \wedge l_1 \neq c \\ maj(c, n-1, L') & : & otherwise \end{cases} \tag{14.19}$$

We also need to define a function, which can verify the result. The idea is that, if the list of votes is empty, the final result is a failure; otherwise, we start the Boyer-Moore algorithm to find a candidate c, then we scan the list again to count the total votes c wins, and verify if this number is not less than the half.

$$majority(L) = \begin{cases} fail & : & L = \Phi \\ c & : & c = maj(l_1, 1, L'), |\{x | x \in L, x = c\}| > \%50|L| \\ fail & : & otherwise \end{cases} \tag{14.20}$$

Below Haskell example code implements this algorithm.

```
majority :: (Eq a) ⇒ [a] → Maybe a
```

[5] We actually uses the ANSI C style. The C++ template is only used to generalize the type of the element

```
majority [] = Nothing
majority (x:xs) = let m = maj x 1 xs in verify m (x:xs)

maj c n [] = c
maj c n (x:xs) | c == x = maj c (n+1) xs
               | n == 0 = maj x 1 xs
               | otherwise = maj c (n-1) xs

verify m xs = if 2 * (length $ filter (==m) xs) > length xs
              then Just m else Nothing
```

Maximum sum of sub vector

Jon Bentley presents another interesting puzzle which can be solved by using quite similar idea in [4]. The problem is to find the maximum sum of sub vector. For example in the following array, The sub vector {19, -12, 1, 9, 18} yields the biggest sum 35.

3	-13	19	-12	1	9	18	-16	15	-15

Note that it is only required to output the value of the maximum sum. If all the numbers are positive, the answer is definitely the sum of all. Another special case is that all numbers are negative. We define the maximum sum is 0 for an empty sub vector.

Of course we can find the answer with brute-force, by calculating all sums of sub vectors and picking the maximum. Such naive method is typical quadratic.

```
1: function MAX-SUM(A)
2:     m ← 0
3:     for i ← 1 to |A| do
4:         s ← 0
5:         for j ← i to |A| do
6:             s ← s + A[j]
7:             m ← MAX(m, s)
8:     return m
```

The brute force algorithm does not reuse any information in previous search. Similar with Boyer-Moore majority vote algorithm, we can record the maximum sum end to the position where we are scanning. Of course we also need record the biggest sum found so far. The following figure illustrates this idea and the invariant during scan.

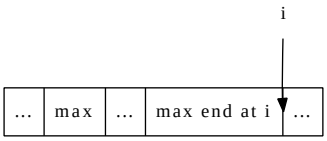


Figure 14.11: Invariant during scan.

At any time when we scan to the i-th position, the max sum found so far is recorded as A. At the same time, we also record the biggest sum end at i as B. Note that A and B may not be the same, in fact, we always maintain $B \leq A$. and when B becomes greater than A by adding with the next element, we update A

with this new value. When B becomes negative, this happens when the next element is a negative number, we reset it to 0. The following tables illustrated the steps when we scan the example vector $\{3, -13, 19, -12, 1, 9, 18, -16, 15, -15\}$.

max sum	max end at i	list to be scan
0	0	$\{3, -13, 19, -12, 1, 9, 18, -16, 15, -15\}$
3	3	$\{-13, 19, -12, 1, 9, 18, -16, 15, -15\}$
3	0	$\{19, -12, 1, 9, 18, -16, 15, -15\}$
19	19	$\{-12, 1, 9, 18, -16, 15, -15\}$
19	7	$\{1, 9, 18, -16, 15, -15\}$
19	8	$\{9, 18, -16, 15, -15\}$
19	17	$\{18, -16, 15, -15\}$
35	35	$\{-16, 15, -15\}$
35	19	$\{15, -15\}$
35	34	$\{-15\}$
35	19	$\{\}$

This algorithm can be described as below.

```
1: function MAX-SUM(V)
2:     A ← 0, B ← 0
3:     for i ← 1 to |V| do
4:         B ← MAX(B + V[i], 0)
5:         A ← MAX(A, B)
```

It is trivial to implement this linear time algorithm, that we skip the details here.

This algorithm can also be defined in functional approach. Instead of mutating variables, we use accumulator to record A and B. In order to search the maximum sum of list L, we call the below function with $max_{sum}(0, 0, L)$.

$$max_{sum}(A, B, L) = \begin{cases} A & : & L = \Phi \\ max_{sum}(A', B', L') & : & otherwise \end{cases} \tag{14.21}$$

Where

$$B' = max(l_1 + B, 0)$$
$$A' = max(A, B')$$

Below Haskell example code implements this algorithm.

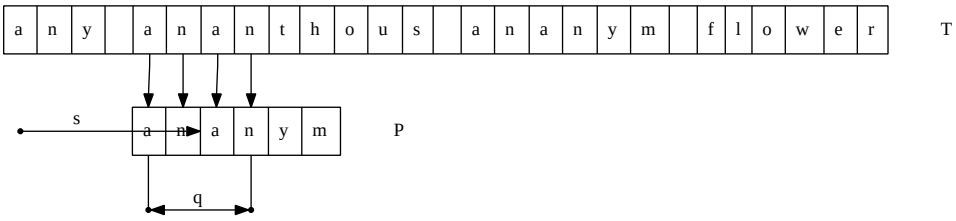
```
maxsum = msum 0 0 where
  msum a _ [] = a
  msum a b (x:xs) = let b' = max (x+b) 0
                        a' = max a b'
                    in msum a' b' xs
```

KMP

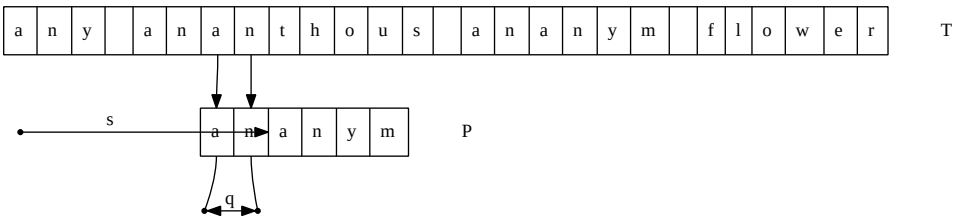
String matching is another important type of searching. Almost all the software editors are equipped with tools to find string in the text. In chapters about Trie, Patricia, and suffix tree, we have introduced some powerful data structures which can help to search string. In this section, we introduce another two string matching algorithms all based on information reusing.

Some programming environments provide built-in string search tools, however, most of them are brute-force solution including 'strstr' function in ANSI

C standard library, 'find' in C++ standard template library, 'indexOf' in Java Development Kit etc. Figure 14.12 illustrate how such character-by-character comparison process works.



(a) The offset $s = 4$, after matching $q = 4$ characters, the 5th mismatch.



(b) Move $s = 4 + 2 = 6$, directly.

Figure 14.12: Match 'ananym' in 'any ananthous ananym flower'.

Suppose we search a pattern P in text T, as shown in figure 14.12 (a), at offset $s = 4$, the process examines every character in P and T to check if they are same. It successfully matches the first 4 characters 'anan'. However, the 5th character in the pattern string is 'y'. It doesn't match the corresponding character in the text, which is 't'.

At this stage, the brute-force solution terminates the attempt, increases s by one to 5, and restart the comparison between 'ananym' and 'nantho...'. Actually, we can increase s not only by one. This is because we have already known that the first four characters 'anan' have been matched, and the failure happens at the 5th position. Observe the two letters prefix 'an' of the pattern string is also a suffix of 'anan' that we have matched so far. A more effective way is to shift s by two but not one, which is shown in figure 14.12 (b). By this means, we reused the information that 4 characters have been matched. This helps us to skip invalid positions as many as possible.

Knuth, Morris and Pratt presented this idea in [9] and developed a novel string matching algorithm. This algorithm is later called as 'KMP', which is consist of the three authors' initials.

For the sake of brevity, we denote the first k characters of text T as T_k. Which means T_k is the k-character prefix of T.

The key point to shift s effectively is to find a function of q, where q is the number of characters matched successfully. For instance, q is 4 in figure 14.12 (a), as the 5th character doesn't match.

Consider what situation we can shift s more than 1. As shown in figure 14.13, if we can shift the pattern P ahead, there must exist k, so that the first k characters are as same as the last k characters of P_q. In other words, the prefix P_k is suffix of P_q.

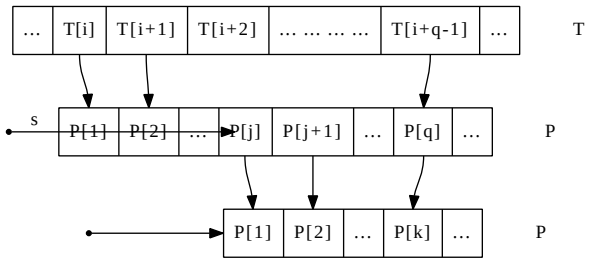


Figure 14.13: P_k is both prefix of P_q and suffix of P_q.

It's possible that there is no such a prefix that is the suffix at the same time. If we treat empty string as both the prefix and the suffix of any others, there must be at least one solution that $k = 0$. It's also quite possible that there are multiple k satisfy. To avoid missing any possible matching positions, we have to find the biggest k. We can define a *prefix function* $\pi(q)$ which tells us where we can fallback if the $(q+1)$-th character does not match [2].

$$\pi(q) = max\{k | k < q \wedge P_k \sqsupset P_q\} \tag{14.22}$$

Where $\sqsupset$ is read as 'is suffix of'. For instance, $A \sqsupset B$ means A is suffix of B. This function is used as the following. When we match pattern P against text T from offset s, If it fails after matching q characters, we next look up $\pi(q)$ to get a fallback q', and retry to compare $P[q']$ with the previous unmatched character. Based on this idea, the core algorithm of KMP can be described as the following.

```
1:  function KMP(T, P)
2:      n ← |T|, m ← |P|
3:      build prefix function π from P
4:      q ← 0                          ▷ How many characters have been matched so far.
5:      for i ← 1 to n do
6:          while q > 0 ∧ P[q + 1] ≠ T[i] do
7:              q ← π(q)
8:          if P[q + 1] = T[i] then
9:              q ← q + 1
10:         if q = m then
11:             found one solution at i − m
12:             q ← π(q)                   ▷ look for next solution
```

Although the definition of prefix function $\pi(q)$ is given in equation (14.22), realizing it blindly by finding the longest suffix isn't effective. Actually we can use the idea of information reusing again to build the prefix function.

The trivial edge case is that, the first character doesn't match. In this case the longest prefix, which is also the suffix is definitely empty, so $\pi(1) = k = 0$. We record the longest prefix as P_k. In this edge case $P_k = P_0$ is the empty string.

After that, when we scan at the q-th character in the pattern string P, we hold the invariant that the prefix function values $\pi(i)$ for i in $\{1, 2, ..., q-1\}$ have already been recorded, and P_k is the longest prefix which is also the suffix of P_{q-1}. As shown in figure 14.14, if $P[q] = P[k+1]$, A bigger k than before

is found, we can increase the maximum of k by one; otherwise, if they are not same, we can use $\pi(k)$ to fallback to a shorter prefix $P_{k'}$ where $k' = \pi(k)$, and check if the next character after this new prefix is same as the q-th character. We need repeat this step until either k becomes zero (which means only empty string satisfies), or the q-th character matches.

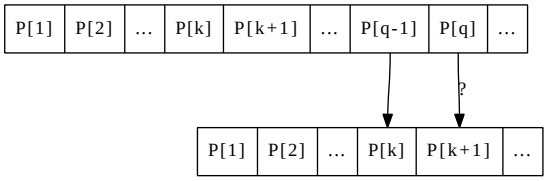


Figure 14.14: P_k is suffix of P_{q-1}, $P[q]$ and $P[k+1]$ are compared.

Realizing this idea gives the KMP prefix building algorithm.

1: **function** BUILD-PREFIX-FUNCTION(P)
2: $m \leftarrow |P|, k \leftarrow 0$
3: $\pi(1) \leftarrow 0$
4: **for** $q \leftarrow 2$ to m **do**
5: **while** $k > 0 \wedge P[q] \neq P[k+1]$ **do**
6: $k \leftarrow \pi(k)$
7: **if** $P[q] = P[k+1]$ **then**
8: $k \leftarrow k + 1$
9: $\pi(q) \leftarrow k$
10: **return** π

The following table lists the steps of building prefix function for pattern string 'ananym'. Note that the k in the table actually means the maximum k satisfies equation (14.22).

q	P_q	k	P_k
1	a	0	""
2	an	0	""
3	ana	1	a
4	anan	2	an
5	anany	0	""
6	ananym	0	""

Translating the KMP algorithm to Python gives the below example code.

```
def kmp_match(w, p):
    n = len(w)
    m = len(p)
    fallback = fprefix(p)
    k = 0 # how many elements have been matched so far.
    res = []
    for i in range(n):
        while k > 0 and p[k] != w[i]:
            k = fallback[k] #fall back
        if p[k] == w[i]:
            k = k + 1
        if k == m:
            res.append(i+1-m)
```

```
            k = fallback[k-1] # look for next
    return res

def fprefix(p):
    m = len(p)
    t = [0]*m  # fallback table
    k = 0
    for i in range(2, m):
        while k>0 and p[i-1] != p[k]:
            k = t[k-1] #fallback
        if p[i-1] == p[k]:
            k = k + 1
        t[i] = k
    return t
```

The KMP algorithm builds the prefix function for the pattern string as a kind of pre-processing before the search. Because of this, it can reuse as much information of the previous matching as possible.

The amortized performance of building the prefix function is $O(m)$. This can be proved by using potential method as in [2]. Using the similar method, it can be proved that the matching algorithm itself is also linear. Thus the total performance is $O(m+n)$ at the expense of the $O(m)$ space to record the prefix function table.

It seems that varies pattern string would affect the performance of KMP. Considering the case that we are finding pattern string 'aaa...a' of length m in a string 'aaa...a' of length n. All the characters are same, when the last character in the pattern is examined, we can only fallback by 1, and this 1 character fallback repeats until it falls back to zero. Even in this extreme case, KMP algorithm still holds its linear performance (why?). Please try to consider more cases such as $P = aaaa...b$, $T = aaaa...a$ and so on.

Purely functional KMP algorithm

It is not easy to realize KMP matching algorithm in purely functional manner. The imperative algorithm represented so far intensely uses array to record prefix function values. Although it is possible to utilize sequence like structure in purely functional settings, such sequence is typically implemented with finger tree. Unlike native arrays, finger tree needs logarithm time for random accessing[6].

Richard Bird presents a formal program deduction to KMP algorithm by using fold fusion law in chapter 17 of [1]. In this section, we show how to develop purely functional KMP algorithm step by step from a brute-force prefix function creation method.

Both text string and pattern are represented as singly linked-list in purely functional settings. During the scan process, these two lists are further partitioned, every one is broken into two parts. As shown in figure 14.15, The first j characters in the pattern string have been matched. $T[i+1]$ and $P[j+1]$ will be compared next. If they are same, we need append the character to the matched

[6]Again, we don't use native array, even it is supported in some functional programming environments like Haskell.

part. However, since strings are essentially singly linked list, such appending is proportion to j.

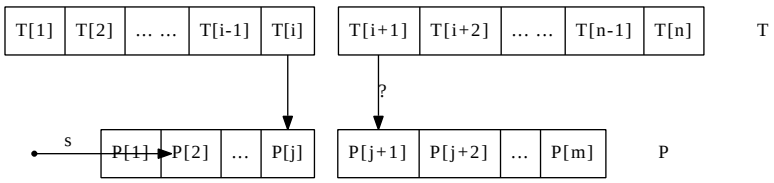


Figure 14.15: P_k is suffix of P_{q-1}, $P[q]$ and $P[k+1]$ are compared.

Denote the first i characters as T_p, which means the prefix of T, the rest characters as T_s for suffix; Similarly, the first j characters as P_p, and the rest as P_s; Denote the first character of T_s as t, the first character of P_s as p. We have the following 'cons' relationship.

$$\begin{aligned} T_s &= cons(t, T'_s) \\ P_s &= cons(p, P'_s) \end{aligned}$$

If $t = p$, note the following updating process is bound to linear time.

$$\begin{aligned} T'_p &= T_p \cup \{t\} \\ P'_p &= P_p \cup \{p\} \end{aligned}$$

We've introduced a method in the chapter about purely functional queue, which can solve this problem. By using a pair of front and rear list, we can turn the linear time appending to constant time linking. The key point is to represent the prefix part in reverse order.

$$\begin{aligned} T = T_p \cup T_s &= reverse(reverse(T_p)) \cup T_s = reverse(\overleftarrow{T_p}) \cup T_s \\ P = P_p \cup P_s &= reverse(reverse(P_p)) \cup P_s = reverse(\overleftarrow{P_p}) \cup P_s \end{aligned} \tag{14.23}$$

The idea is to using pair $(\overleftarrow{T_p}, T_s)$ and $(\overleftarrow{P_p}, P_s)$ instead. With this change, the if $t = p$, we can update the prefix part fast in constant time.

$$\begin{aligned} \overleftarrow{T'_p} &= cons(t, \overleftarrow{T_p}) \\ \overleftarrow{P'_p} &= cons(p, \overleftarrow{P_p}) \end{aligned} \tag{14.24}$$

The KMP matching algorithm starts by initializing the success prefix parts to empty strings as the following.

$$search(P, T) = kmp(\pi, (\Phi, P)(\Phi, T)) \tag{14.25}$$

Where π is the prefix function we explained before. The core part of KMP

algorithm, except for the prefix function building, can be defined as below.

$$kmp(\pi, (\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) = \begin{cases} \{|\overleftarrow{T_p}|\} & : & P_s = \Phi \wedge T_s = \Phi \\ \Phi & : & P_s \neq \Phi \wedge T_s = \Phi \\ \{|\overleftarrow{T_p}\} \cup kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) & : & P_s = \Phi \wedge T_s \neq \Phi \\ kmp(\pi, (\overleftarrow{P'_p}, P'_s), (\overleftarrow{T'_p}, T'_s)) & : & t = p \\ kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T'_p}, T'_s)) & : & t \neq p \wedge \overleftarrow{P_p} = \Phi \\ kmp(\pi, \pi(\overleftarrow{P_p}, P_s), (\overleftarrow{T_p}, T_s)) & : & t \neq p \wedge \overleftarrow{P_p} \neq \Phi \end{cases} \tag{14.26}$$

The first clause states that, if the scan successfully ends to both the pattern and text strings, we get a solution, and the algorithm terminates. Note that we use the right position in the text string as the matching point. It's easy to use the left position by subtracting with the length of the pattern string. For sake of brevity, we switch to right position in functional solutions.

The second clause states that if the scan arrives at the end of text string, while there are still rest of characters in the pattern string haven't been matched, there is no solution. And the algorithm terminates.

The third clause states that, if all the characters in the pattern string have been successfully matched, while there are still characters in the text haven't been examined, we get a solution, and we fallback by calling prefix function π to go on searching other solutions.

The fourth clause deals with the case, that the next character in pattern string and text are same. In such case, the algorithm advances one character ahead, and recursively performs searching.

If the the next characters are not same and this is the first character in the pattern string, we just need advance to next character in the text, and try again. Otherwise if this isn't the first character in the pattern, we call prefix function π to fallback, and try again.

The brute-force way to build the prefix function is just to follow the definition equation (14.22).

$$\pi(\overleftarrow{P_p}, P_s) = (\overleftarrow{P'_p}, P'_s) \tag{14.27}$$

where

$$\begin{aligned} P'_p &= longest(\{s | s \in prefixes(P_p), s \sqsupset P_p\}) \\ P'_s &= P - P'_p \end{aligned}$$

Every time when calculate the fallback position, the algorithm naively enumerates all prefixes of P_p, checks if it is also the suffix of P_p, and then pick the longest one as result. Note that we reuse the subtraction symbol here for list differ operation.

There is a tricky case which should be avoided. Because any string itself is both its prefix and suffix. Say $P_p \sqsubset P_p$ and $P_p \sqsupset P_p$. We shouldn't enumerate P_p as a candidate prefix. One solution of such prefix enumeration can be realized as the following.

$$prefixes(L) = \begin{cases} \{\Phi\} & : & L = \Phi \vee |L| = 1 \\ cons(\Phi, map(\lambda_s \cdot cons(l_1, s), prefixes(L'))) & : & otherwise \end{cases} \tag{14.28}$$

Below Haskell example program implements this version of string matching algorithm.

```
kmpSearch1 ptn text = kmpSearch' next ([], ptn) ([], text)

kmpSearch' _ (sp, []) (sw, []) = [length sw]
kmpSearch' _ _ (_, []) = []
kmpSearch' f (sp, []) (sw, ws) = length sw : kmpSearch' f (f sp []) (sw, ws)
kmpSearch' f (sp, (p:ps)) (sw, (w:ws))
    | p == w = kmpSearch' f ((p:sp), ps) ((w:sw), ws)
    | otherwise = if sp ==[] then kmpSearch' f (sp, (p:ps)) ((w:sw), ws)
                  else kmpSearch' f (f sp (p:ps)) (sw, (w:ws))

next sp ps = (sp', ps') where
    prev = reverse sp
    prefix = longest [xs | xs ← inits prev, xs `isSuffixOf` prev]
    sp' = reverse prefix
    ps' = (prev ++ ps) \\ prefix
    longest = maximumBy (compare `on` length)

inits [] = [[]]
inits [_] = [[]]
inits (x:xs) = [] : (map (x:) $ inits xs)
```

This version does not only perform poorly, but it is also complex. We can simplify it a bit. Observing the KMP matching is a scan process from left to the right of the text, it can be represented with folding (refer to Appendix A for detail). Firstly, we can augment each character with an index for folding like below.

$$zip(T, \{1, 2, ...\}) \tag{14.29}$$

Zipping the text string with infinity natural numbers gives list of pairs. For example, text string 'The quick brown fox jumps over the lazy dog' turns into (T, 1), (h, 2), (e, 3), ... (o, 42), (g, 43).

The initial state for folding contains two parts, one is the pair of pattern (P_p, P_s), with prefix starts from empty, and the suffix is the whole pattern string (Φ, P). For illustration purpose only, we revert back to normal pairs but not $(\overleftarrow{P_p}, P_s)$ notation. It can be easily replaced with reversed form in the finalized version. This is left as exercise to the reader. The other part is a list of positions, where the successful matching are found. It starts from empty list. After the folding finishes, this list contains all solutions. What we need is to extract this list from the final state. The core KMP search algorithm is simplified like this.

$$kmp(P, T) = snd(fold(search, ((\Phi, P), \Phi), zip(T, \{1, 2, ...\}))) \tag{14.30}$$

The only 'black box' is the *search* function, which takes a state, and a pair of character and index, and it returns a new state as result. Denote the first character in P_s as p and the rest characters as P'_s ($P_s = cons(p, P'_s)$), we have

the following definition.

$$search(((P_p, P_s), L), (c, i)) = \begin{cases} ((P_p \cup p, P_s'), L \cup \{i\}) & : \quad p = c \wedge P_s' = \Phi \\ ((P_p \cup p, P_s'), L) & : \quad p = c \wedge P_s' \neq \Phi \\ ((P_p, P_s), L) & : \quad P_p = \Phi \\ search((\pi(P_p, P_s), L), (c, i)) & : \quad otherwise \end{cases} \tag{14.31}$$

If the first character in P_s matches the current character c during scan, we need further check if all the characters in the pattern have been examined, if so, we successfully find a solution, This position i in list L is recorded; Otherwise, we advance one character ahead and go on. If p does not match c, we need fallback for further retry. However, there is an edge case that we can't fallback any more. P_p is empty in this case, and we need do nothing but keep the current state.

The prefix-function π developed so far can also be improved a bit. Since we want to find the longest prefix of P_p, which is also suffix of it, we can scan from right to left instead. For any non empty list L, denote the first element as l_1, and all the rest except for the first one as L', define a function $init(L)$, which returns all the elements except for the last one as below.

$$init(L) = \begin{cases} \Phi & : \quad |L| = 1 \\ cons(l_1, init(L')) & : \quad otherwise \end{cases} \tag{14.32}$$

Note that this function can not handle empty list. The idea of scan from right to left for P_p is first check if $init(P_p) \sqsupset P_p$, if yes, then we are done; otherwise, we examine if $init(init(P_p))$ is OK, and repeat this till the left most. Based on this idea, the prefix-function can be modified as the following.

$$\pi(P_p, P_s) = \begin{cases} (P_p, P_s) & : \quad P_p = \Phi \\ fallback(init(P_p), cons(last(P_p), P_s)) & : \quad otherwise \end{cases} \tag{14.33}$$

Where

$$fallback(A, B) = \begin{cases} (A, B) & : \quad A \sqsupset P_p \\ (init(A), cons(last(A), B)) & : \quad otherwise \end{cases} \tag{14.34}$$

Note that fallback always terminates because empty string is suffix of any string. The $last(L)$ function returns the last element of a list, it is also a linear time operation (refer to Appendix A for detail). However, it's constant operation if we use $\overleftarrow{P_p}$ approach. This improved prefix-function is bound to linear time. It is still quite slower than the imperative algorithm which can look up prefix-function in constant $O(1)$ time. The following Haskell example program implements this minor improvement.

```
failure ([], ys) = ([], ys)
failure (xs, ys) = fallback (init xs) (last xs:ys) where
    fallback as bs | as `isSuffixOf` xs = (as, bs)
                   | otherwise = fallback (init as) (last as:bs)

kmpSearch ws txt = snd $ foldl f (([], ws), []) (zip txt [1..]) where
```

```
f (p@(xs, (y:ys)), ns) (x, n) | x == y = if ys==[] then ((xs++[y], ys), ns++[n])
                                         else ((xs++[y], ys), ns)
                              | xs == [] = (p, ns)
                              | otherwise = f (failure p, ns) (x, n)
f (p, ns) e = f (failure p, ns) e
```

The bottleneck is that we can not use native array to record prefix functions in purely functional settings. In fact the prefix function can be understood as a state transform function. It transfer from one state to the other according to the matching is success or fail. We can abstract such state changing as a tree. In environment supporting algebraic data type, Haskell for example, such state tree can be defined like below.

```
data State a = E | S a (State a) (State a)
```

A state is either empty, or contains three parts: the current state, the new state if match fails, and the new state if match succeeds. Such definition is quite similar to the binary tree. We can call it 'left-fail, right-success' tree. The state we are using here is (P_p, P_s).

Similar as imperative KMP algorithm, which builds the prefix function from the pattern string, the state transforming tree can also be built from the pattern. The idea is to build the tree from the very beginning state (Φ, P), with both its children empty. We replace the left child with a new state by calling π function defined above, and replace the right child by advancing one character ahead. There is an edge case, that when the state transfers to (P, Φ), we can not advance any more in success case, such node only contains child for failure case. The build function is defined as the following.

$$build((P_p, P_s), \Phi, \Phi) = \begin{cases} build(\pi(P_p, P_s), \Phi, \Phi) & : \quad P_s = \Phi \\ build((P_p, P_s), L, R) & : \quad otherwise \end{cases} \tag{14.35}$$

Where

$$L = build(\pi(P_p, P_s), \Phi, \Phi)$$
$$R = build((P_s \cup \{p\}, P'_s), \Phi, \Phi))$$

The meaning of p and P'_s are as same as before, that p is the first character in P_s, and P'_s is the rest characters. The most interesting point is that the build function will never stop. It endless build a infinite tree. In strict programming environment, calling this function will freeze. However, in environments support lazy evaluation, only the nodes have to be used will be created. For example, both Haskell and Scheme/Lisp are capable to construct such infinite state tree. In imperative settings, it is typically realized by using pointers which links to ancestor of a node.

Figure 14.16 illustrates such an infinite state tree for pattern string 'ananym'. Note that the right most edge represents the case that the matching continuously succeed for all characters. After that, since we can't match any more, so the right sub-tree is empty. Base on this fact, we can define a auxiliary function to test if a state indicates the whole pattern is successfully matched.

$$match((P_p, P_s), L, R) = \begin{cases} True & : \quad P_s = \Phi \\ False & : \quad otherwise \end{cases} \tag{14.36}$$

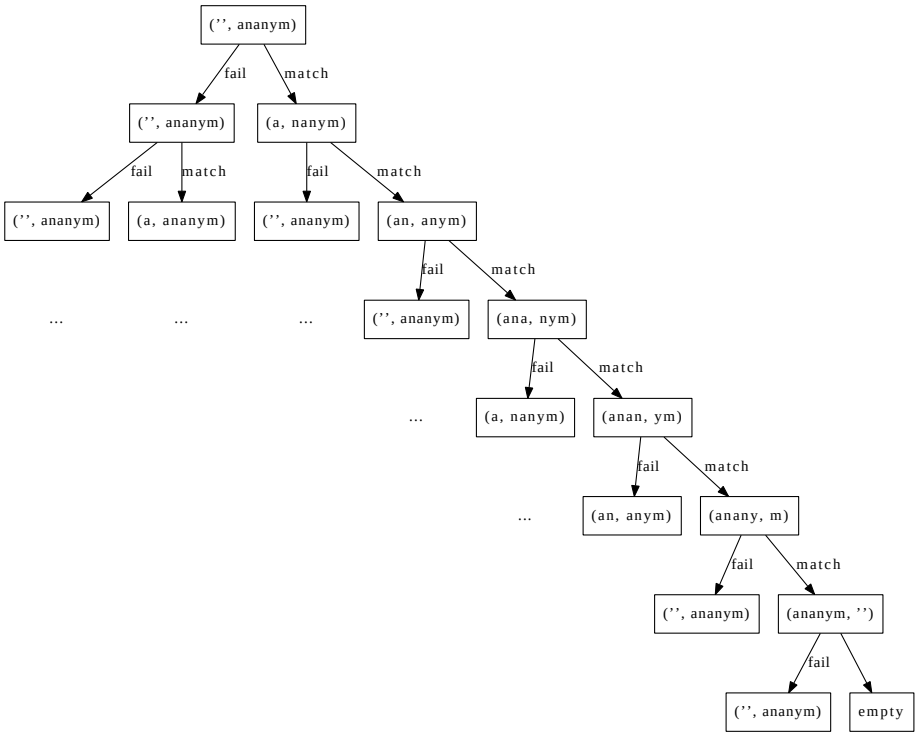


Figure 14.16: The infinite state tree for pattern ‘ananym’.

With the help of state transform tree, we can realize KMP algorithm in an automaton manner.

$$kmp(P, T) = snd(fold(search, (Tr, []), zip(T, \{1, 2, ...\}))) \tag{14.37}$$

Where the tree $Tr = build((\Phi, P), \Phi, \Phi)$ is the infinite state transform tree. Function *search* utilizes this tree to transform the state according to match or fail. Denote the first character in P_s as p, the rest characters as P'_s, and the matched positions found so far as A.

$$search((((P_p, P_s), L, R), A), (c, i)) = \begin{cases} (R, A \cup \{i\}) & : & p = c \wedge match(R) \\ (R, A) & : & p = c \wedge \neg match(R) \\ ((((P_p, P_s), L, R), A) & : & P_p = \Phi \\ search((L, A), (c, i)) & : & otherwise \end{cases} \tag{14.38}$$

The following Haskell example program implements this algorithm.

```
data State a = E | S a (State a) (State a) -- state, ok-state, fail-state
               deriving (Eq, Show)

build :: (Eq a)⇒State ([a], [a]) → State ([a], [a])
build (S s@(xs, []) E E) = S s (build (S (failure s) E E)) E
build (S s@(xs, (y:ys)) E E) = S s l r where
    l = build (S (failure s) E E) -- fail state
    r = build (S (xs++[y], ys) E E)

matched (S (_, []) _ _) = True
matched _ = False

kmpSearch3 :: (Eq a) ⇒ [a] → [a] → [Int]
kmpSearch3 ws txt = snd $ foldl f (auto, []) (zip txt [1..]) where
    auto = build (S ([], ws) E E)
    f (s@(S (xs, ys) l r), ns) (x, n)
        | [x] `isPrefixOf` ys = if matched r then (r, ns++[n])
                                else (r, ns)
        | xs == [] = (s, ns)
        | otherwise = f (l, ns) (x, n)
```

The bottle-neck is that the state tree building function calls π to fallback. While current definition of π isn't effective enough, because it enumerates all candidates from right to the left every time.

Since the state tree is infinite, we can adopt some common treatment for infinite structures. One good example is the Fibonacci series. The first two Fibonacci numbers are defined as 0 and 1; the rest Fibonacci numbers can be obtained by adding the previous two numbers.

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned} \tag{14.39}$$

Thus the Fibonacci numbers can be list one by one as the following

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_2 &= F_1 + F_0 \\ F_3 &= F_2 + F_1 \\ &... \end{aligned} \tag{14.40}$$

We can collect all numbers in both sides, and define $F = \{0, 1, F_1, F_2, ...\}$, Thus we have the following equation.

$$\begin{aligned} F = & \{0, 1, F_1 + F_0, F_2 + F_1, ...\} \\ = & \{0, 1\} \cup \{x + y | x \in \{F_0, F_1, F_2, ...\}, y \in \{F_1, F_2, F_3, ...\}\} \\ = & \{0, 1\} \cup \{x + y | x \in F, y \in F'\} \end{aligned} \tag{14.41}$$

Where $F' = tail(F)$ is all the Fibonacci numbers except for the first one. In environments support lazy evaluation, like Haskell for instance, this definition can be expressed like below.

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

The recursive definition for infinite Fibonacci series indicates an idea which can be used to get rid of the fallback function π. Denote the state transfer tree as T, we can define the transfer function when matching a character on this tree as the following.

$$trans(T, c) = \begin{cases} root & : & T = \Phi \\ R & : & T = ((P_p, P_s), L, R), c = p \\ trans(L, c) & : & otherwise \end{cases} \tag{14.42}$$

If we match a character against empty node, we transfer to the root of the tree. We'll define the root later soon. Otherwise, we compare if the character c is as same as the first character p in P_s. If they match, then we transfer to the right sub tree for this success case; otherwise, we transfer to the left sub tree for fail case.

With transfer function defined, we can modify the previous tree building function accordingly. This is quite similar to the previous Fibonacci series definition.

$$build(T, (P_p, P_s)) = ((P_p, P_s), T, build(trans(T, p), (P_p \cup \{p\}, P'_s)))$$

The right hand of this equation contains three parts. The first one is the state that we are matching (P_p, P_s); If the match fails, Since T itself can handle any fail case, we use it directly as the left sub tree; otherwise we recursive build the right sub tree for success case by advancing one character ahead, and calling transfer function we defined above.

However, there is an edge case which has to be handled specially, that if P_s is empty, which indicates a successful match. As defined above, there isn't right sub tree any more. Combining these cases gives the final building function.

$$build(T, (P_p, P_s)) = \begin{cases} ((P_p, P_s), T, \Phi) & : & P_s = \Phi \\ ((P_p, P_s), T, build(trans(T, p), (P_p \cup \{p\}, P'_s))) & : & otherwise \end{cases} \tag{14.43}$$

The last brick is to define the root of the infinite state transfer tree, which initializes the building.

$$root = build(\Phi, (\Phi, P)) \tag{14.44}$$

And the new KMP matching algorithm is modified with this root.

$$kmp(P, T) = snd(fold(trans, (root, []), zip(T, \{1, 2, ...\}))) \tag{14.45}$$

The following Haskell example program implements this final version.

```
kmpSearch ws txt = snd $ foldl tr (root, []) (zip txt [1..]) where
    root = build' E ([], ws)
    build' fails (xs, []) = S (xs, []) fails E
    build' fails s@(xs, (y:ys)) = S s fails succs where
        succs = build' (fst (tr (fails, []) (y, 0))) (xs++[y], ys)
    tr (E, ns) _ = (root, ns)
    tr ((S (xs, ys) fails succs), ns) (x, n)
        | [x] `isPrefixOf` ys = if matched succs then (succs, ns++[n]) else (succs, ns)
        | otherwise = tr (fails, ns) (x, n)
```

Figure 14.17 shows the first 4 steps when search 'anaym' in text 'anal'. Since the first 3 steps all succeed, so the left sub trees of these 3 states are not actually constructed. They are marked as '?'. In the fourth step, the match fails, thus the right sub tree needn't be built. On the other hand, we must construct the left sub tree, which is on top of the result of $trans(right(right(right(T))), n)$, where function $right(T)$ returns the right sub tree of T. This can be further expanded according to the definition of building and state transforming functions till we get the concrete state $((a, nanym), L, R)$. The detailed deduce process is left as exercise to the reader.

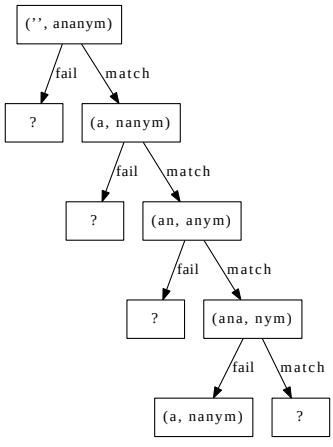


Figure 14.17: On demand construct the state transform tree when searching 'ananym' in text 'anal'.

This algorithm depends on the lazy evaluation critically. All the states to be transferred are built on demand. So that the building process is amortized

$O(m)$, and the total performance is amortized $O(n + m)$. Readers can refer to [1] for detailed proof of it.

It's worth of comparing the final purely functional and the imperative algorithms. In many cases, we have expressive functional realization, however, for KMP matching algorithm, the imperative approach is much simpler and more intuitive. This is because we have to mimic the raw array by a infinite state transfer tree.

Boyer-Moore

Boyer-Moore string matching algorithm is another effective solution invited in 1977 [10]. The idea of Boyer-Moore algorithm comes from the following observation.

The bad character heuristics

When attempt to match the pattern, even if there are several characters from the left are same, it fails if the last one does not match, as shown in figure 14.18. What's more, we wouldn't find a match even if we slide the pattern down by 1, or 2. Actually, the length of the pattern 'ananym' is 5, the last character is 'm', however, the corresponding character in the text is 'h'. It does not appear in the pattern at all. We can directly slide the pattern down by 5.

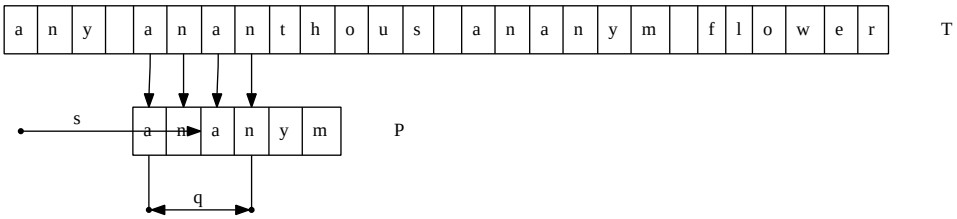


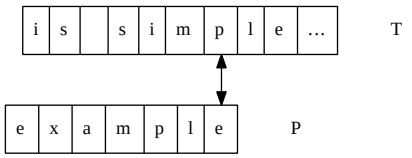
Figure 14.18: Since character 'h' doesn't appear in the pattern, we wouldn't find a match even if we slide the pattern down less than the length of the pattern.

This leads to *the bad-character rule*. We can do a pre-processing for the pattern. If the character set of the text is already known, we can find all characters which don't appear in the pattern string. During the later scan process, as long as we find such a bad character, we can immediately slide the pattern down by its length. The question is what if the unmatched character does appear in the pattern? While, in order not to miss any potential matches, we have to slide down the pattern to check again. This is shown as in the figure 14.19

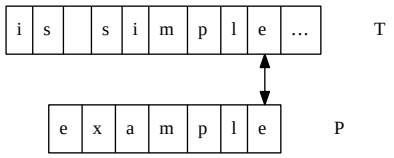
It's quite possible that the unmatched character appears in the pattern more than one position. Denote the length of the pattern as $|P|$, the character appears in positions $p_1, p_2, ..., p_i$. In such case, we take the right most one to avoid missing any matches.

$$s = |P| - p_i \tag{14.46}$$

Note that the shifting length is 0 for the last position in the pattern according to the above equation. Thus we can skip it in realization. Another important



(a) The last character in the pattern 'e' doesn't match 'p'. However, 'p' appears in the pattern.



(b) We have to slide the pattern down by 2 to check again.

Figure 14.19: Slide the pattern if the unmatched character appears in the pattern.

point is that since the shifting length is calculated against the position aligned with the last character in the pattern string, (we deduce it from $|P|$), no matter where the mismatching happens when we scan from right to the left, we slide down the pattern string by looking up the bad character table with the one in the text aligned with the last character of the pattern. This is shown in figure 14.20.

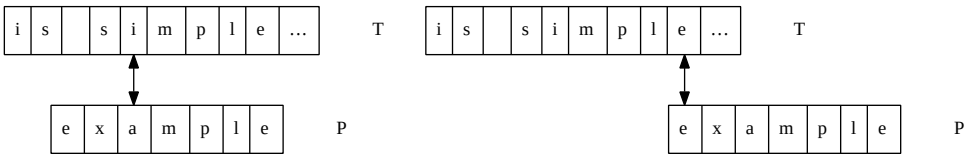


Figure 14.20: Even the mismatching happens in the middle, between char 'i' and 'a', we look up the shifting value with character 'e', which is 6 (calculated from the first 'e', the second 'e' is skipped to avoid zero shifting).

There is a good result in practice, that only using the bad-character rule leads to a simple and fast string matching algorithm, called Boyer-Moore-Horspool algorithm [11].

1: **procedure** BOYER-MOORE-HORSPOOL(T, P)
2: **for** $\forall c \in \Sigma$ **do**
3: $\pi[c] \leftarrow |P|$
4: **for** $i \leftarrow 1$ to $|P| - 1$ **do** ▷ Skip the last position
5: $\pi[P[i]] \leftarrow |P| - i$
6: $s \leftarrow 0$
7: **while** $s + |P| \leq |T|$ **do**

```
8:      i ← |P|
9:      while i ≥ 1 ∧ P[i] = T[s + i] do                ▷ scan from right
10:         i ← i − 1
11:     if i < 1 then
12:         found one solution at s
13:         s ← s + 1                                   ▷ go on finding the next
14:     else
15:         s ← s + π[T[s + |P|]]
```

The character set is denoted as Σ, we first initialize all the values of sliding table π as the length of the pattern string $|P|$. After that we process the pattern from left to right, update the sliding value. If a character appears multiple times in the pattern, the latter value, which is on the right hand, will overwrite the previous value. We start the matching scan process by aligning the pattern and the text string from the very left. However, for every alignment s, we scan from the right to the left until either there is unmatched character or all the characters in the pattern have been examined. The latter case indicates that we've found a match; while for the former case, we look up π to slide the pattern down to the right.

The following example Python code implements this algorithm accordingly.

```
def bmh_match(w, p):
    n = len(w)
    m = len(p)
    tab = [m for _ in range(256)]  # table to hold the bad character rule.
    for i in range(m-1):
        tab[ord(p[i])] = m - 1 - i
    res = []
    offset = 0
    while offset + m ≤ n:
        i = m - 1
        while i ≥ 0 and p[i] == w[offset+i]:
            i = i - 1
        if i < 0:
            res.append(offset)
            offset = offset + 1
        else:
            offset = offset + tab[ord(w[offset + m - 1])]
    return res
```

The algorithm firstly takes about $O(|\Sigma| + |P|)$ time to build the sliding table. If the character set size is small, the performance is dominated by the pattern and the text. There is definitely the worst case that all the characters in the pattern and text are same, e.g. searching 'aa...a' (m of 'a', denoted as a^m) in text 'aa.....a' (n of 'a', denoted as a^n). The performance in the worst case is $O(mn)$. This algorithm performs well if the pattern is long, and there are constant number of matching. The result is bound to linear time. This is as same as the best case of full Boyer-Moore algorithm which will be explained next.

The good suffix heuristics

Consider searching pattern 'abbabab' in text 'bbbababbabab...' like figure 14.21. By using the bad-character rule, the pattern will be slided by two.

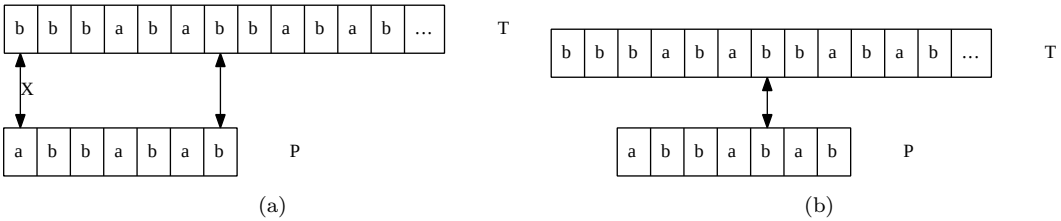


Figure 14.21: According to the bad-character rule, the pattern is slided by 2, so that the next 'b' is aligned.

Actually, we can do better than this. Observing that before the unmatched point, we have already successfully matched 6 characters 'bbabab' from right to the left. Since 'ab', which is the prefix of the pattern is also the suffix of what we matched so far, we can directly slide the pattern to align this suffix as shown in figure 14.22.

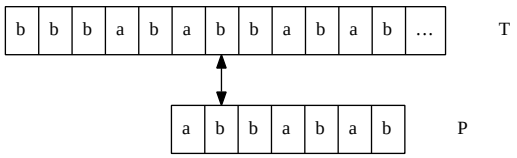


Figure 14.22: As the prefix 'ab' is also the suffix of what we've matched, we can slide down the pattern to a position so that 'ab' are aligned.

This is quite similar to the pre-processing of KMP algorithm, However, we can't always skip so many characters. Consider the following example as shown in figure 14.23. We have matched characters 'bab' when the unmatch happens. Although the prefix 'ab' of the pattern is also the suffix of 'bab', we can't slide the pattern so far. This is because 'bab' appears somewhere else, which starts from the 3rd character of the pattern. In order not to miss any potential matching, we can only slide the pattern by two.

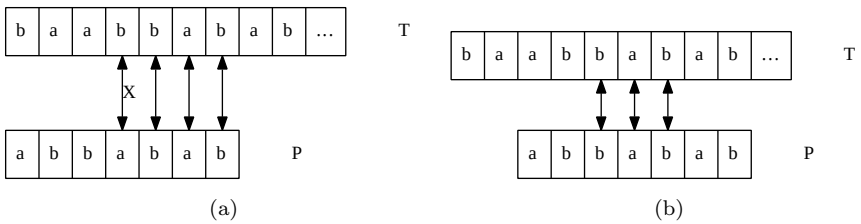
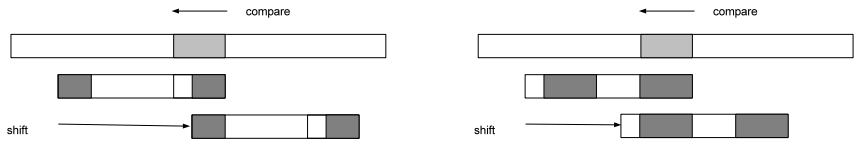


Figure 14.23: We've matched 'bab', which appears somewhere else in the pattern (from the 3rd to the 5th character). We can only slide down the pattern by 2 to avoid missing any potential matching.

The above situation forms the two cases of *the good-suffix rule*, as shown in figure 14.24.



(a) Case 1, Only a part of the matching suffix occurs as a prefix of the pattern.

(b) Case 2, The matching suffix occurs some where else in the pattern.

Figure 14.24: The light gray section in the text represents the characters have been matched; The dark gray parts indicate the same content in the pattern.

Both cases in good suffix rule handle the situation that there are multiple characters have been matched from right. We can slide the pattern to the right if any of the the following happens.

- Case 1 states that if a part of the matching suffix occurs as a prefix of the pattern, and the matching suffix doesn't appear in any other places in the pattern, we can slide the pattern to the right to make this prefix aligned;
- Case 2 states that if the matching suffix occurs some where else in the pattern, we can slide the pattern to make the right most occurrence aligned.

Note that in the scan process, we should apply case 2 first whenever it is possible, and then examine case 1 if the whole matched suffix does not appears in the pattern. Observe that both cases of the good-suffix rule only depend on the pattern string, a table can be built by pre-process the pattern for further looking up.

For the sake of brevity, we denote the suffix string from the i-th character of P as $\overline{P_i}$. That $\overline{P_i}$ is the sub-string $P[i]P[i+1]...P[m]$.

For case 1, we can check every suffix of P, which includes $\overline{P_m}$, $\overline{P_{m-1}}$, $\overline{P_{m-2}}$, ..., $\overline{P_2}$ to examine if it is the prefix of P. This can be achieved by a round of scan from right to the left.

For case 2, we can check every prefix of P includes P_1, P_2, ..., P_{m-1} to examine if the longest suffix is also a suffix of P. This can be achieved by another round of scan from left to the right.

```
1: function GOOD-SUFFIX(P)
2:     m ← |P|
3:     π_s ← {0, 0, ..., 0}                    ▷ Initialize the table of length m
4:     l ← 0                                   ▷ The last suffix which is also prefix of P
5:     for i ← m − 1 down-to 1 do              ▷ First loop for case 1
6:         if P̄_i ⊏ P then                     ▷ ⊏ means 'is prefix of'
7:             l ← i
```

```
8:          π_s[i] ← l
9:      for i ← 1 to m do                          ▷ Second loop for case 2
10:         s ← Suffix-Length(P_i)
11:         if s ≠ 0 ∧ P[i − s] ≠ P[m − s] then
12:             π_s[m − s] ← m − i
13:     return π_s
```

This algorithm builds the good-suffix heuristics table π_s. It first checks every suffix of P from the shortest to the longest. If the suffix $\overline{P_i}$ is also the prefix of P, we record this suffix, and use it for all the entries until we find another suffix $\overline{P_j}$, $j < i$, and it is also the prefix of P.

After that, the algorithm checks every prefix of P from the shortest to the longest. It calls the function Suffix-Length(P_i), to calculate the length of the longest suffix of P_i, which is also suffix of P. If this length s isn't zero, which means there exists a sub-string of s, that appears as the suffix of the pattern. It indicates that case 2 happens. The algorithm overwrites the s-th entry from the right of the table π_s. Note that to avoid finding the same occurrence of the matched suffix, we test if $P[i - s]$ and $P[m - s]$ are same.

Function Suffix-Length is designed as the following.

```
1: function Suffix-Length(P_i)
2:     m ← |P|
3:     j ← 0
4:     while P[m − j] = P[i − j] ∧ j < i do
5:         j ← j + 1
6:     return j
```

The following Python example program implements the good-suffix rule.

```
def good_suffix(p):
    m = len(p)
    tab = [0 for _ in range(m)]
    last = 0
    # first loop for case 1
    for i in range(m-1, 0, -1): # m-1, m-2, ◦ .., 1
        if is_prefix(p, i):
            last = i
        tab[i - 1] = last
    # second loop for case 2
    for i in range(m):
        slen = suffix_len(p, i)
        if slen != 0 and p[i - slen] != p[m - 1 - slen]:
            tab[m - 1 - slen] = m - 1 - i
    return tab

# test if p[i..m-1] 'is prefix of' p
def is_prefix(p, i):
    for j in range(len(p) - i):
        if p[j] != p [i+j]:
            return False
    return True

# length of the longest suffix of p[..i], which is also a suffix of p
def suffix_len(p, i):
```

```
    m = len(p)
    j = 0
    while p[m - 1 - j] == p[i - j] and j < i:
        j = j + 1
    return j
```

It's quite possible that both the bad-character rule and the good-suffix rule can be applied when the unmatch happens. The Boyer-Moore algorithm compares and picks the bigger shift so that it can find the solution as quick as possible. The bad-character rule table can be explicitly built as below

1: **function** BAD-CHARACTER(P)
2: **for** $\forall c \in \Sigma$ **do**
3: $\pi_b[c] \leftarrow |P|$
4: **for** $i \leftarrow 1$ to $|P| - 1$ **do**
5: $\pi_b[P[i]] \leftarrow |P| - i$
6: **return** π_b

The following Python program implements the bad-character rule accordingly.

```
def bad_char(p):
    m = len(p)
    tab = [m for _ in range(256)]
    for i in range(m-1):
        tab[ord(p[i])] = m - 1 - i
    return tab
```

The final Boyer-Moore algorithm firstly builds the two rules from the pattern, then aligns the pattern to the beginning of the text and scans from right to the left for every alignment. If any unmatch happens, it tries both rules, and slides the pattern with the bigger shift.

1: **function** BOYER-MOORE(T, P)
2: $n \leftarrow |T|, m \leftarrow |P|$
3: $\pi_b \leftarrow$ BAD-CHARACTER(P)
4: $\pi_s \leftarrow$ GOOD-SUFFIX(P)
5: $s \leftarrow 0$
6: **while** $s + m \leq n$ **do**
7: $i \leftarrow m$
8: **while** $i \geq 1 \wedge P[i] = T[s + i]$ **do**
9: $i \leftarrow i - 1$
10: **if** $i < 1$ **then**
11: found one solution at s
12: $s \leftarrow s + 1$ ▷ go on finding the next
13: **else**
14: $s \leftarrow s + max(\pi_b[T[s + m]], \pi_s[i])$

Here is the example implementation of Boyer-Moore algorithm in Python.

```
def bm_match(w, p):
    n = len(w)
    m = len(p)
    tab1 = bad_char(p)
    tab2 = good_suffix(p)
    res = []
```

```
    offset = 0
    while offset + m ≤ n:
        i = m - 1
        while i ≥ 0 and p[i] == w[offset + i]:
            i = i - 1
        if i < 0:
            res.append(offset)
            offset = offset + 1
        else:
            offset = offset + max(tab1[ord(w[offset + m - 1])], tab2[i])
    return res
```

The Boyer-Moore algorithm published in original paper is bound to $O(n+m)$ in worst case only if the pattern doesn't appear in the text [10]. Knuth, Morris, and Pratt proved this fact in 1977 [12]. However, when the pattern appears in the text, as we shown above, Boyer-Moore performs $O(nm)$ in the worst case.

Richard birds shows a purely functional realization of Boyer-Moore algorithm in chapter 16 in [1]. We skipped it in this book.

Exercise 14.2

- Proof that Boyer-Moore majority vote algorithm is correct.

- Given a list, find the element occurs most. Are there any divide and conqueror solutions? Are there any divide and conqueror data structures, such as map can be used?

- Bentley presents a divide and conquer algorithm to find the maximum sum in $O(n \log n)$ time in [4]. The idea is to split the list at the middle point. We can recursively find the maximum sum in the first half and second half; However, we also need to find maximum sum cross the middle point. The method is to scan from the middle point to both ends as the following.

```
1: function MAX-SUM(A)
2:     if A = Φ then
3:         return 0
4:     else if |A| = 1 then
5:         return MAX(0, A[1])
6:     else
7:         m ← ⌊|A|/2⌋
8:         a ← MAX-FROM(REVERSE(A[1...m]))
9:         b ← MAX-FROM(A[m + 1...|A|])
10:        c ← MAX-SUM(A[1...m])
11:        d ← MAX-SUM(A[m + 1...|A|)
12:        return MAX(a + b, c, d)

13: function MAX-FROM(A)
14:     sum ← 0, m ← 0
15:     for i ← 1 to |A| do
16:         sum ← sum + A[i]
17:         m ← MAX(m, sum)
18:     return m
```

It's easy to deduce the time performance is $T(n) = 2T(n/2) + O(n)$. Implement this algorithm in your favorite programming language.

- Explain why KMP algorithm perform in linear time even in the seemed 'worst' case.
- Implement the purely functional KMP algorithm by using reversed P_p to avoid the linear time appending operation.
- Deduce the state of the tree $left(right(right(right(T))))$ when searching 'ananym' in text 'anal'.

14.3 Solution searching

One interesting thing that computer programming can offer is solving puzzles. In the early phase of classic artificial intelligent, people developed many methods to search for solutions. Different from the sequence searching and string matching, the solution doesn't obviously exist among a candidates set. It typically need construct the solution while trying varies of attempts. Some problems are solvable, while others are not. Among the solvable problems, not all of them just have one unique solution. For example, a maze may have multiple ways out. People sometimes need search for the best one.

14.3.1 DFS and BFS

DFS and BFS stand for deep-first search and breadth-first search. They are typically introduced as graph algorithms in textbooks. Graph is a comprehensive topic which is hard to be covered in this elementary book. In this section, we'll show how to use DFS and BFS to solve some real puzzles without formal introduction about the graph concept.

Maze

Maze is a classic and popular puzzle. Maze is amazing to both kids and adults. Figure 14.25 shows an example maze. There are also real maze gardens can be found in parks for fun. In the late 1990s, maze-solving games were quite often hold in robot mouse competition all over the world.

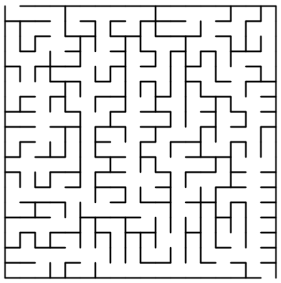


Figure 14.25: A maze

There are multiple methods to solve maze puzzle. We'll introduce an effective, yet not the best one in this section. There are some well known sayings about how to find the way out in maze, while not all of them are true.

For example, one method states that, wherever you have multiple ways, always turn right. This doesn't work as shown in figure 14.26. The obvious solution is first to go along the top horizontal line, then turn right, and keep going ahead at the 'T' section. However, if we always turn right, we'll endless loop around the inner big block.

Figure 14.26: It leads to loop way if always turns right.

This example tells us that the decision when there are multiple choices matters the solution. Like the fairy tale we read in our childhood, we can take some bread crumbs in a maze. When there are multiple ways, we can simply select one, left a piece of bread crumbs to mark this attempt. If we enter a died end, we go back to the last place where we've made a decision by back-tracking the bread crumbs. Then we can alter to another way.

At any time, if we find there have been already bread crumbs left, it means we have entered a loop, we must go back and try different ways. Repeat these try-and-check steps, we can either find the way out, or give the 'no solution' fact. In the later case, we back-track to the start point.

One easy way to describe a maze, is by a $m \times n$ matrix, each element is either 0 or 1, which indicates if there is a way at this cell. The maze illustrated in figure 14.26 can be defined as the following matrix.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{matrix}$$

Given a start point $s = (i, j)$, and a goal $e = (p, q)$, we need find all solutions, that are the paths from s to e.

There is an obviously recursive exhaustive search method. That in order to find all paths from s to e, we can check all connected points to s, for every such point k, we recursively find all paths from k to e. This method can be illustrated as the following.

- Trivial case, if the start point s is as same as the target point e, we are done;

- Otherwise, for every connected point k to s, recursively find the paths from k to e; If e can be reached via k, put section s-k in front of each path between k and e.

However, we have to left 'bread crumbs' to avoid repeatedly trying the same attempts. This is because otherwise in the recursive case, we start from s, find a connected point k, then we further try to find paths from k to e. Since s is connected to k as well, so in the next recursion, we'll try to find paths from s to e again. It turns to be the very same origin problem, and we are trapped in infinite recursions.

Our solution is to initialize an empty list, use it to record all the points we've visited so far. For every connected point, we look up the list to examine if it has already been visited. We skip all the visited candidates and only try those new ones. The corresponding algorithm can be defined like this.

$$solveMaze(m, s, e) = solve(s, \{\Phi\}) \tag{14.47}$$

Where m is the matrix which defines a maze, s is the start point, and e is the end point. Function $solve$ is defined in the context of $solveMaze$, so that the maze and the end point can be accessed. It can be realized recursively like what we described above[7].

$$solve(s, P) = \begin{cases} \{\{s\} \cup p | p \in P\} & : \quad s = e \\ concat(\{ \quad solve(s', \{\{s\} \cup p | p \in P\}) | \\ \qquad s' \in adj(s), \neg visited(s')\}) & : \quad otherwise \end{cases} \tag{14.48}$$

Note that P also serves as an accumulator. Every connected point is recorded in all the possible paths to the current position. But they are stored in reversed order, that is the newly visited point is put to the head of all the lists, and the starting point is the last one. This is because the appending operation is linear ($O(n)$, where n is the number of elements stored in a list), while linking to the head is just constant time. We can output the result in correct order by reversing all possible solutions in equation (14.47)[8]:

$$solveMaze(m, s, e) = map(reverse, solve(s, \{\Phi\})) \tag{14.49}$$

We need define functions $adj(p)$ and $visited(p)$, which finds all the connected points to p, and tests if point p has been visited respectively. Two points are connected if and only if they are next cells horizontally or vertically in the maze matrix, and both have zero value.

$$adj((x, y)) = \{(x', y') | \quad (x', y') \in \{(x-1, y), (x+1, y), (x, y-1), (x, y+1)\}, \\ 1 \leq x' \leq M, 1 \leq y' \leq N, m_{x'y'} = 0\} \tag{14.50}$$

Where M and N are the widths and heights of the maze.

Function $visited(p)$ examines if point p has been recorded in any lists in P.

$$visited(p, P) = \exists path \in P, p \in path \tag{14.51}$$

[7] Function *concat* can flatten a list of lists. For example. $concat(\{\{a, b, c\}, \{x, y, z\}\}) = \{a, b, c, x, y, z\}$. Refer to appendix A for detail.

[8] the detailed definition of *reverse* can be found in the appendix A.

The following Haskell example code implements this algorithm.

```
solveMaze m from to = map reverse $ solve from [[]] where
    solve p paths | p == to = map (p:) paths
                  | otherwise = concat [solve p' (map (p:) paths) |
                                        p' ← adjacent p,
                                        not $ visited p' paths]
    adjacent (x, y) = [(x', y') |
                       (x', y') ← [(x-1, y), (x+1, y), (x, y-1), (x, y+1)],
                       inRange (bounds m) (x', y'),
                       m ! (x', y') == 0]
    visited p paths = any (p `elem`) paths
```

For a maze defined as matrix like below example, all the solutions can be given by this program.

```
mz = [[0, 0, 1, 0, 1, 1],
      [1, 0, 1, 0, 1, 1],
      [1, 0, 0, 0, 0, 0],
      [1, 1, 0, 1, 1, 1],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 1, 1, 0]]

maze  = listArray ((1,1), (6, 6)) ∘ concat

solveMaze (maze mz) (1,1) (6, 6)
```

As we mentioned, this is a style of 'exhaustive search'. It recursively searches all the connected points as candidates. In a real maze solving game, a robot mouse competition for instance, it's enough to just find a route. We can adapt to a method close to what described at the beginning of this section. The robot mouse always tries the first connected point, and skip the others until it gets stuck. We need some data structure to store the 'bread crumbs', which help to remember the decisions being made. As we always attempt to find the way on top of the latest decision, it is the last-in, first-out manner. A stack can be used to realize it.

At the very beginning, only the starting point s is stored in the stack. we pop it out, find, for example, points a, and b, are connected to s. We push the two possible paths: $\{a, s\}$ and $\{b, s\}$ to the stack. Next we pop $\{a, s\}$ out, and examine connected points to a. Then all the paths with 3 steps will be pushed back. We repeat this process. At anytime, each element stored in the stack is a path, from the starting point to the farthest place can arrive in the reversed order. This can be illustrated in figure 14.27.

The stack can be realized with a list. The latest option is picked from the head, and the new candidates are also added to the head. The maze puzzle can be solved by using such a list of paths:

$$solveMaze'(m, s, e) = reverse(solve'(\{\{s\}\})) \tag{14.52}$$

As we are searching the first, but not all the solutions, *map* isn't used here. When the stack is empty, it means that we've tried all the options and failed to find a way out. There is no solution; otherwise, the top option is popped, expanded with all the adjacent points which haven't been visited before, and pushed back to the stack. Denote the stack as S, if it isn't empty, the top

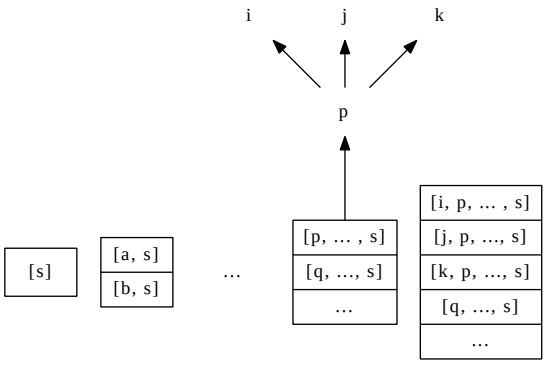


Figure 14.27: The stack is initialized with a singleton list of the starting point s. s is connected with point a and b. Paths $\{a, s\}$ and $\{b, s\}$ are pushed back. In some step, the path ended with point p is popped. p is connected with points i, j, and k. These 3 points are expanded as different options and pushed back to the stack. The candidate path ended with q won't be examined unless all the options above fail.

element is s_1, and the new stack after the top being popped as S'. s_1 is a list of points represents path P. Denote the first point in this path as p_1, and the rest as P'. The solution can be formalized as the following.

$$solve'(S) = \begin{cases} \Phi & : S = \Phi \\ s_1 & : s_1 = e \\ solve'(S') & : C = \{c | c \in adj(p_1), c \notin P'\} = \Phi \\ solve'(\{\{p\} \cup P | p \in C\} \cup S) & : otherwise, C \neq \Phi \end{cases} \tag{14.53}$$

Where the *adj* function is defined above. This updated maze solution can be implemented with the below example Haskell program [9].

```
dfsSolve m from to = reverse $ solve [[from]] where
    solve [] = []
    solve (c@(p:path):cs)
        | p == to = c -- stop at the first solution
        | otherwise = let os = filter (`notElem` path) (adjacent p) in
                      if os == []
                      then solve cs
                      else solve ((map (:c) os) ++ cs)
```

It's quite easy to modify this algorithm to find all solutions. When we find a path in the second clause, instead of returning it immediately, we record it and go on checking the rest memorized options in the stack till until the stack becomes empty. We left it as an exercise to the reader.

The same idea can also be realized imperatively. We maintain a stack to store all possible paths from the starting point. In each iteration, the top option path is popped, if the farthest position is the end point, a solution is found; otherwise, all the adjacent, not visited yet points are appended as new paths and pushed

[9] The same code of *adjacent* function is skipped

back to the stack. This is repeated till all the candidate paths in the stacks are checked.

We use the same notation to represent the stack S. But the paths will be stored as arrays instead of list in imperative settings as the former is more effective. Because of this the starting point is the first element in the path array, while the farthest reached place is the right most element. We use p_n to represent $\textsc{Last}(P)$ for path P. The imperative algorithm can be given as below.

```
1: function SOLVE-MAZE(m, s, e)
2:     S ← Φ
3:     PUSH(S, {s})
4:     L ← Φ                                          ▷ the result list
5:     while S ≠ Φ do
6:         P ← POP(S)
7:         if e = p_n then
8:             ADD(L, P)
9:         else
10:            for ∀p ∈ ADJACENT(m, p_n) do
11:                if p ∉ P then
12:                    PUSH(S, P ∪ {p})
13:    return L
```

The following example Python program implements this maze solving algorithm.

```
def solve(m, src, dst):
    stack = [[src]]
    s = []
    while stack != []:
        path = stack.pop()
        if path[-1] == dst:
            s.append(path)
        else:
            for p in adjacent(m, path[-1]):
                if not p in path:
                    stack.append(path + [p])
    return s

def adjacent(m, p):
    (x, y) = p
    ds = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    ps = []
    for (dx, dy) in ds:
        x1 = x + dx
        y1 = y + dy
        if 0 <= x1 and x1 < len(m[0]) and
           0 <= y1 and y1 < len(m) and m[y][x] == 0:
            ps.append((x1, y1))
    return ps
```

And the same maze example given above can be solved by this program like the following.

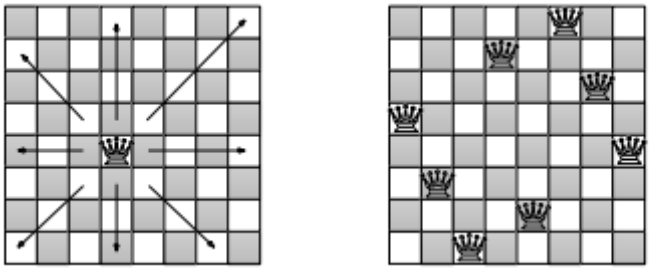
```
mz = [[0, 0, 1, 0, 1, 1],
```

```
        [1, 0, 1, 0, 1, 1],
        [1, 0, 0, 0, 0, 0],
        [1, 1, 0, 1, 1, 1],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, 1, 0]]

solve(mz, (0, 0), (5,5))
```

It seems that in the worst case, there are 4 options (up, down, left, and right) at each step, each option is pushed to the stack and eventually examined during backtracking. Thus the complexity is bound to $O(4^n)$. The actual time won't be so large because we filtered out the places which have been visited before. In the worst case, all the reachable points are visited exactly once. So the time is bound to $O(n)$, where n is the number of points connected in total. As a stack is used to store candidate solutions, the space complexity is $O(n^2)$.

Eight queens puzzle

The eight queens puzzle is also a famous problem. Although cheese has very long history, this puzzle was first published in 1848 by Max Bezzel[13]. Queen in the cheese game is quite powerful. It can attack any other pieces in the same row, column and diagonal at any distance. The puzzle is to find a solution to put 8 queens in the board, so that none of them attack each other. Figure 14.28 (a) illustrates the places can be attacked by a queen and 14.28 (b) shows a solution of 8 queens puzzle.



(a) A queen piece. (b) An example solution

Figure 14.28: The eight queens puzzle.

It's obviously that the puzzle can be solved by brute-force, which takes P_{64}^8 times. This number is about 4×10^{10}. It can be easily improved by observing that, no two queens can be in the same row, and each queen must be put on one column between 1 to 8. Thus we can represent the arrangement as a permutation of $\{1, 2, 3, 4, 5, 6, 7, 8\}$. For instance, the arrangement $\{6, 2, 7, 1, 3, 5, 8, 4\}$ means, we put the first queen at row 1, column 6, the second queen at row 2 column 2, ..., and the last queen at row 8, column 4. By this means, we need only examine $8! = 40320$ possibilities.

We can find better solutions than this. Similar to the maze puzzle, we put queens one by one from the first row. For the first queen, there are 8 options, that we can put it at one of the eight columns. Then for the next queen, we

again examine the 8 candidate columns. Some of them are not valid because those positions will be attacked by the first queen. We repeat this process, for the i-th queen, we examine the 8 columns in row i, find which columns are safe. If none column is valid, it means all the columns in this row will be attacked by some queen we've previously arranged, we have to backtrack as what we did in the maze puzzle. When all the 8 queens are successfully put to the board, we find a solution. In order to find all the possible solutions, we need record it and go on to examine other candidate columns and perform back tracking if necessary. This process terminates when all the columns in the first row have been examined. The below equation starts the search.

$$solve(\{\Phi\}, \Phi) \tag{14.54}$$

In order to manage the candidate attempts, a stack S is used as same as in the maze puzzle. The stack is initialized with one empty element. And a list L is used to record all possible solutions. Denote the top element in the stack as s_1. It's actually an intermediate state of assignment, which is a partial permutation of 1 to 8. after pops s_1, the stack becomes S'. The $solve$ function can be defined as the following.

$$solve(S, L) = \begin{cases} L & : \quad S = \Phi \\ solve(S', \{s_1\} \cup L) & : \quad |s_1| = 8 \\ solve(\left\{ \begin{array}{ll} \{i\} \cup s_1 | & i \in [1, 8], \\ & i \notin s_1, \\ & safe(i, s_1) \end{array} \right\} \cup S', L) & : \quad otherwise \end{cases} \tag{14.55}$$

If the stack is empty, all the possible candidates have been examined, it's not possible to backtrack any more. L has been accumulated all found solutions and returned as the result; Otherwise, if the length of the top element in the stack is 8, a valid solution is found. We add it to L, and go on finding other solutions; If the length is less than 8, we need try to put the next queen. Among all the columns from 1 to 8, we pick those not already occupied by previous queens (through the $i \notin s_1$ clause), and must not be attacked in diagonal direction (through the $safe$ predication). The valid assignments will be pushed to the stack for the further searching.

Function $safe(x, C)$ detects if the assignment of a queen in position x will be attacked by other queens in C in diagonal direction. There are 2 possible cases, 45° and 135° directions. Since the row of this new queen is $y = 1 + |C|$, where $|C|$ is the length of C, the $safe$ function can be defined as the following.

$$safe(x, C) = \forall (c, r) \in zip(reverse(C), \{1, 2, ...\}), |x - c| \neq |y - r| \tag{14.56}$$

Where zip takes two lists, and pairs every elements in them to a new list. Thus If $C = \{c_{i-1}, c_{i-2}, ..., c_2, c_1\}$ represents the column of the first $i - 1$ queens has been assigned, the above function will check list of pairs $\{(c_1, 1), (c_2, 2), ..., (c_{i-1}, i - 1)\}$ with position (x, y) forms any diagonal lines.

Translating this algorithm into Haskell gives the below example program.

```
solve = dfsSolve [[]] [] where
    dfsSolve [] s = s
```

```
dfsSolve (c:cs) s
         | length c == 8 = dfsSolve cs (c:s)
         | otherwise = dfsSolve ([(x:c) | x ← [1..8] \\ c,
                                  not $ attack x c] ++ cs) s
attack x cs = let y = 1 + length cs in
              any (λ(c, r) → abs(x - c) == abs(y - r)) $
                  zip (reverse cs) [1..]
```

Observing that the algorithm is tail recursive, it's easy to transform it into imperative realization. Instead of using list, we use array to represent queens assignment. Denote the stack as S, and the possible solutions as A. The imperative algorithm can be described as the following.

```
1: function SOLVE-QUEENS
2:     S ← {Φ}
3:     L ← Φ                                          ▷ The result list
4:     while S ≠ Φ do
5:         A ← POP(S)                          ▷ A is an intermediate assignment
6:         if |A| = 8 then
7:             ADD(L, A)
8:         else
9:             for i ← 1 to 8 do
10:                if VALID(i, A) then
11:                    PUSH(S, A ∪ {i})
12:    return L
```

The stack is initialized with the empty assignment. The main process repeatedly pops the top candidate from the stack. If there are still queens left, the algorithm examines possible columns in the next row from 1 to 8. If a column is safe, that it won't be attacked by any previous queens, this column will be appended to the assignment, and pushed back to the stack. Different from the functional approach, since array, but not list, is used, we needn't reverse the solution assignment any more.

Function VALID checks if column x is safe with previous queens put in A. It filters out the columns have already been occupied, and calculates if any diagonal lines are formed with existing queens.

```
1: function VALID(x, A)
2:     y ← 1 + |A|
3:     for i ← 1 to |A| do
4:         if x = i ∨ |y − i| = |x − A[i]| then
5:             return False
6:     return True
```

The following Python example program implements this imperative algorithm.

```
def solve():
    stack = [[]]
    s = []
    while stack != []:
        a = stack.pop()
        if len(a) == 8:
            s.append(a)
        else:
```

```
            for i in range(1, 9):
                if valid(i, a):
                    stack.append(a+[i])
    return s

def valid(x, a):
    y = len(a) + 1
    for i in range(1, y):
        if x == a[i-1] or abs(y - i) == abs(x - a[i-1]):
            return False
    return True
```

Although there are 8 optional columns for each queen, not all of them are valid and thus further expanded. Only those columns haven't been occupied by previous queens are tried. The algorithm only examines 15720, which is far less than $8^8 = 16777216$, possibilities [13].

It's quite easy to extend the algorithm, so that it can solve N queens puzzle, where $N \geq 4$. However, the time cost increases fast. The backtrack algorithm is just slightly better than the one permuting the sequence of 1 to 8 (which is bound to $o(N!)$). Another extension to the algorithm is based on the fact that the chess board is square, which is symmetric both vertically and horizontally. Thus a solution can generate other solutions by rotating and flipping. These aspects are left as exercises to the reader.

Peg puzzle

I once received a puzzle of the leap frogs. It said to be homework for 2nd grade student in China. As illustrated in figure 14.29, there are 6 frogs in 7 stones. Each frog can either hop to the next stone if it is not occupied, or leap over one frog to another empty stone. The frogs on the left side can only move to the right, while the ones on the right side can only move to the left. These rules are described in figure 14.30

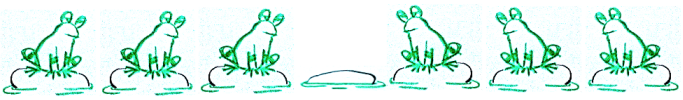


Figure 14.29: The leap frogs puzzle.

The goal of this puzzle is to arrange the frogs to jump according to the rules, so that the positions of the 3 frogs on the left are finally exchange with the ones on the right. If we denote the frog on the left as 'A', on the right as 'B', and the empty stone as 'O'. The puzzle is to find a solution to transform from 'AAAOBBB' to 'BBBOAAA'.

This puzzle is just a special form of the peg puzzles. The number of pegs is not limited to 6. it can be 8 or other bigger even numbers. Figure 14.31 shows some variants.

(a) Jump to the next stone (b) Jump over to the right (c) Jump over to the left

Figure 14.30: Moving rules.

(a) Solitaire (b) Hop over (c) Draught board

Figure 14.31: Variants of the peg puzzles from http://home.comcast.net/ stegmann/jumping.htm

We can solve this puzzle by programing. The idea is similar to the 8 queens puzzle. Denote the positions from the left most stone as 1, 2, ..., 7. In ideal cases, there are 4 options to arrange the move. For example when start, the frog on 3rd stone can hop right to the empty stone; symmetrically, the frog on the 5th stone can hop left; Alternatively, the frog on the 2nd stone can leap right, while the frog on the 6th stone can leap left.

We can record the state and try one of these 4 options at every step. Of course not all of them are possible at any time. If get stuck, we can backtrack and try other options.

As we restrict the left side frogs only moving to the right, and the right frogs only moving to the left. The moves are not reversible. There won't be any repetition cases as what we have to deal with in the maze puzzle. However, we still need record the steps in order to print them out finally.

In order to enforce these restriction, let A, O, B in representation 'AAAOBBB' be -1, 0, and 1 respectively. A state L is a list of elements, each element is one of these 3 values. It starts from $\{-1, -1, -1, 0, 1, 1, 1\}$. $L[i]$ access the i-th element, its value indicates if the i-th stone is empty, occupied by a frog from left side, or occupied by a frog from right side. Denote the position of the vacant stone as p. The 4 moving options can be stated as below.

- Leap left: $p < 6$ and $L[p+2] > 0$, swap $L[p] \leftrightarrow L[p+2]$;
- Hop left: $p < 7$ and $L[p+1] > 0$, swap $L[p] \leftrightarrow L[p+1]$;
- Leap right: $p > 2$ and $L[p-2] < 0$, swap $L[p-2] \leftrightarrow L[p]$;
- Hop right: $p > 1$ and $L[p-1] < 0$, swap $L[p-1] \leftrightarrow L[p]$.

Four functions $leap_l(L), hop_l(L), leap_r(L)$ and $hop_r(L)$ are defined accordingly. If the state L does not satisfy the move restriction, these function return L unchanged, otherwise, the changed state L' is returned accordingly.

We can also explicitly maintain a stack S to the attempts as well as the historic movements. The stack is initialized with a singleton list of starting state. The solution is accumulated to a list M, which is empty at the beginning:

$$solve(\{\{-1,-1,-1,0,1,1,1\}\},\Phi) \tag{14.57}$$

As far as the stack isn't empty, we pop one intermediate attempt. If the latest state is equal to $\{1,1,1,0,-1,-1,-1\}$, a solution is found. We append the series of moves till this state to the result list M; otherwise, We expand to next possible state by trying all four possible moves, and push them back to the stack for further search. Denote the top element in the stack S as s_1, and the latest state in s_1 as L. The algorithm can be defined as the following.

$$solve(S,M) = \begin{cases} M & : & S = \Phi \\ solve(S', \{reverse(s_1)\} \cup M) & : & L = \{1,1,1,0,-1,-1,-1\} \\ solve(P \cup S', M) & : & otherwise \end{cases} \tag{14.58}$$

Where P are possible moves from the latest state L:

$$P = \{L' | L' \in \{leap_l(L), hop_l(L), leap_r(L), hop_r(L)\}, L \neq L'\}$$

Note that the starting state is stored as the last element, while the final state is the first. That is the reason why we reverse it when adding to solution list.

Translating this algorithm to Haskell gives the following example program.

```
solve = dfsSolve [[[-1, -1, -1, 0, 1, 1, 1]]] [] where
    dfsSolve [] s = s
    dfsSolve (c:cs) s
             | head c == [1, 1, 1, 0, -1, -1, -1] = dfsSolve cs (reverse c:s)
             | otherwise = dfsSolve ((map (:c) $ moves $ head c) ++ cs) s

moves s = filter (/=s) [leapLeft s, hopLeft s, leapRight s, hopRight s] where
    leapLeft [] = []
    leapLeft (0:y:1:ys) = 1:y:0:ys
    leapLeft (y:ys) = y:leapLeft ys
    hopLeft [] = []
    hopLeft (0:1:ys) = 1:0:ys
    hopLeft (y:ys) = y:hopLeft ys
    leapRight [] = []
    leapRight (-1:y:0:ys) = 0:y:(-1):ys
    leapRight (y:ys) = y:leapRight ys
    hopRight [] = []
    hopRight (-1:0:ys) = 0:(-1):ys
    hopRight (y:ys) = y:hopRight ys
```

Running this program finds 2 symmetric solutions, each takes 15 steps. One solution is list in the below table.

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

Observe that the algorithm is in tail recursive manner, it can also be realized imperatively. The algorithm can be more generalized, so that it solve the puzzles of n frogs on each side. We represent the start state {-1, -1, ..., -1, 0, 1, 1, ..., 1} as s, and the mirrored end state as e.

```
1: function SOLVE(s, e)
2:     S ← {{s}}
3:     M ← Φ
4:     while S ≠ Φ do
5:         s_1 ← POP(S)
6:         if s_1[1] = e then
7:             ADD(M, REVERSE(s_1))
8:         else
9:             for ∀m ∈ MOVES(s_1[1]) do
10:                PUSH(S, {m} ∪ s_1)
11:    return M
```

The possible moves can be also generalized with procedure MOVES to handle arbitrary number of frogs. The following Python program implements this solution.

```
def solve(start, end):
    stack = [[start]]
    s = []
    while stack != []:
        c = stack.pop()
        if c[0] == end:
            s.append(reversed(c))
        else:
            for m in moves(c[0]):
                stack.append([m]+c)
    return s

def moves(s):
    ms = []
    n = len(s)
    p = s.index(0)
```

```
    if p < n - 2 and s[p+2] > 0:
        ms.append(swap(s, p, p+2))
    if p < n - 1 and s[p+1] > 0:
        ms.append(swap(s, p, p+1))
    if p > 1 and s[p-2] < 0:
        ms.append(swap(s, p, p-2))
    if p > 0 and s[p-1] < 0:
        ms.append(swap(s, p, p-1))
    return ms

def swap(s, i, j):
    a = s[:]
    (a[i], a[j]) = (a[j], a[i])
    return a
```

For 3 frogs in each side, we know that it takes 15 steps to exchange them. It's interesting to examine the table that how many steps are needed along with the number of frogs in each side. Our program gives the following result.

number of frogs	1	2	3	4	5	...
number of steps	3	8	15	24	35	...

It seems that the number of steps are all square numbers minus one. It's natural to guess that the number of steps for n frogs in one side is $(n+1)^2 - 1$. Actually we can prove it is true.

Compare to the final state and the start state, each frog moves ahead $n+1$ stones in its opposite direction. Thus total $2n$ frogs move $2n(n+1)$ stones. Another important fact is that each frog on the left has to meet every one on the right one time. And leap will happen when meets. Since the frog moves two stones ahead by leap, and there are total n^2 meets happen, so that all these meets cause moving $2n^2$ stones ahead. The rest moves are not leap, but hop. The number of hops are $2n(n+1) - 2n^2 = 2n$. Sum up all n^2 leaps and $2n$ hops, the total number of steps are $n^2 + 2n = (n+1)^2 - 1$.

Summary of DFS

Observe the above three puzzles, although they vary in many aspects, their solutions show quite similar common structures. They all have some starting state. The maze starts from the entrance point; The 8 queens puzzle starts from the empty board; The leap frogs start from the state of 'AAAOBBB'. The solution is a kind of searching, at each attempt, there are several possible ways. For the maze puzzle, there are four different directions to try; For the 8 queens puzzle, there are eight columns to choose; For the leap frogs puzzle, there are four movements of leap or hop. We don't know how far we can go when make a decision, although the final state is clear. For the maze, it's the exit point; For the 8 queens puzzle, we are done when all the 8 queens being assigned on the board; For the leap frogs puzzle, the final state is that all frogs exchanged.

We use a common approach to solve them. We repeatedly select one possible candidate to try, record where we've achieved; If we get stuck, we backtrack and try other options. We are sure by using this strategy, we can either find a solution, or tell that the problem is unsolvable.

Of course there can be some variation, that we can stop when find one answer, or go on searching all the solutions.

If we draw a tree rooted at the starting state, expand it so that every branch stands for a different attempt, our searching process is in a manner, that it searches deeper and deeper. We won't consider any other options in the same depth unless the searching fails so that we've to backtrack to upper level of the tree. Figure 14.32 illustrates the order we search a state tree. The arrow indicates how we go down and backtrack up. The number of the nodes shows the order we visit them.

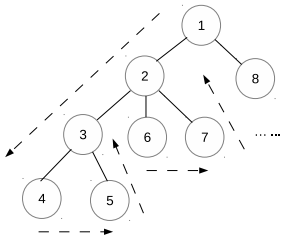


Figure 14.32: Example of DFS search order.

This kind of search strategy is called 'DFS' (Deep-first-search). We widely use it unintentionally. Some programming environments, Prolog for instance, adopt DFS as the default evaluation model. A maze is given by a set of rule base, such as:

```
c(a, b). c(a, e).
c(b, c). c(b, f).
c(e, d), c(e, f).
c(f, c).
c(g, d). c(g, h).
c(h, f).
```

Where predicate $c(X, Y)$ means place X is connected with Y. Note that this is a directed predicate, we can make Y to be connected with X as well by either adding a symmetric rule, or create a undirected predicate. Figure 14.33 shows such a directed graph. Given two places X and Y, Prolog can tell if they are connected by the following program.

```
go(X, X).
go(X, Y) :- c(X, Z), go(Z, Y)
```

This program says that, a place is connected with itself. Given two different places X and Y, if X is connected with Z, and Z is connected with Y, then X is connected with Y. Note that, there might be multiple choices for Z. Prolog selects a candidate, and go on further searching. It only tries other candidates if the recursive searching fails. In that case, Prolog backtracks and tries other alternatives. This is exactly what DFS does.

DFS is quite straightforward when we only need a solution, but don't care if the solution takes the fewest steps. For example, the solution it gives, may not be the shortest path for the maze. We'll see some more puzzles next. They demands to find the solution with the minimum attempts.

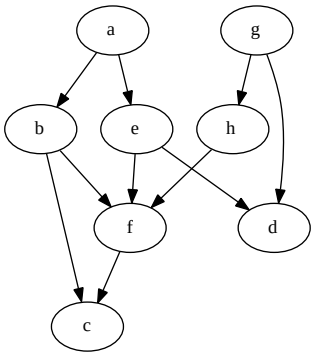


Figure 14.33: A directed graph.

The wolf, goat, and cabbage puzzle

This puzzle says that a farmer wants to cross a river with a wolf, a goat, and a bucket of cabbage. There is a boat. Only the farmer can drive it. But the boat is small. it can only hold one of the wolf, the goat, and the bucket of cabbage with the farmer at a time. The farmer has to pick them one by one to the other side of the river. However, the wolf would eat the goat, and the goat would eat the cabbage if the farmer is absent. The puzzle asks to find the fast solution so that they can all safely go cross the river.

Figure 14.34: The wolf, goat, cabbage puzzle

The key point to this puzzle is that the wolf does not eat the cabbage. The farmer can safely pick the goat to the other side. But next time, no matter if he pick the wolf or the cabbage to cross the river, he has to take one back to avoid the conflict. In order to find the fast the solution, at any time, if the farmer has multiple options, we can examine all of them in parallel, so that these different decisions compete. If we count the number of the times the farmer cross the river without considering the direction, that crossing the river back and forth means 2 times, we are actually checking the complete possibilities after 1 time,

2 times, 3 times, ... When we find a situation, that they all arrive at the other bank, we are done. And this solution wins the competition, which is the fast solution.

The problem is that we can't examine all the possible solutions in parallel ideally. Even with a super computer equipped with many CPU cores, the setup is too expensive to solve such a simple puzzle.

Let's consider a lucky draw game. People blindly pick from a box with colored balls. There is only one black ball, all the others are white. The one who pick the black ball wins the game; Otherwise, he must return the ball to the box and wait for the next chance. In order to be fair enough, we can setup a rule that no one can try the second time before all others have tried. We can line people to a queue. Every time the first guy pick a ball, if he does not win, he then stands at the tail of the queue to wait for the second try. This queue helps to ensure our rule.

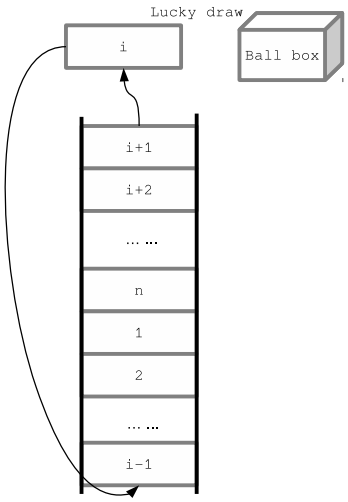


Figure 14.35: A lucky-draw game, the i-th person goes from the queue, pick a ball, then join the queue at tail if he fails to pick the black ball.

We can use the quite same idea to solve our puzzle. The two banks of the river can be represented as two sets A and B. A contains the wolf, the goat, the cabbage and the farmer; while B is empty. We take an element from one set to the other each time. The two sets can't hold conflict things if the farmer is absent. The goal is to exchange the contents of A and B with fewest steps.

We initialize a queue with state $A = \{w, g, c, p\}, B = \Phi$ as the only element. As far as the queue isn't empty, we pick the first element from the head, expand it with all possible options, and put these new expanded candidates to the tail of the queue. If the first element on the head is the final goal, that $A = \Phi, B = \{w, g, c, p\}$, we are done. Figure 14.36 illustrates the idea of this search order. Note that as all possibilities in the same level are examined, there is no need for back-tracking.

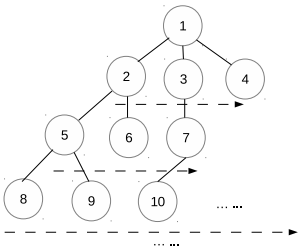


Figure 14.36: Start from state 1, check all possible options 2, 3, and 4 for next step; then all nodes in level 3, ...

There is a simple way to treat the set. A four bits binary number can be used, each bit stands for a thing, for example, the wolf $w = 1$, the goat $g = 2$, the cabbage $c = 4$, and the farmer $p = 8$. That 0 stands for the empty set, 15 stands for a full set. Value 3, solely means there are a wolf and a goat on the river bank. In this case, the wolf will eat the goat. Similarly, value 6 stands for another conflicting case. Every time, we move the highest bit (which is 8), or together with one of the other bits (4 or 2, or 1) from one number to the other. The possible moves can be defined as below.

$$mv(A, B) = \begin{cases} \{(A - 8 - i, B + 8 + i) | i \in \{0, 1, 2, 4\}, i = 0 \vee A \overline{\wedge} i \neq 0\} & : \quad B < 8 \\ \{(A + 8 + i, B - 8 - i) | i \in \{0, 1, 2, 4\}, i = 0 \vee B \overline{\wedge} i \neq 0\} & : \quad Otherwise \end{cases} \tag{14.59}$$

Where $\overline{\wedge}$ is the bitwise-and operation.

the solution can be given by reusing the queue defined in previous chapter. Denote the queue as Q, which is initialed with a singleton list $\{(15, 0)\}$. If Q is not empty, function $DeQ(Q)$ extracts the head element M, the updated queue becomes Q'. M is a list of pairs, stands for a series of movements between the river banks. The first element in $m_1 = (A', B')$ is the latest state. Function $EnQ'(Q, L)$ is a slightly different enqueue operation. It pushes all the possible moving sequences in L to the tail of the queue one by one and returns the updated queue. With these notations, the solution function is defined like below.

$$solve(Q) = \begin{cases} \Phi & : \quad Q = \Phi \\ reverse(M) & : \quad A' = 0 \\ solve(EnQ'(Q', \left\{ \{m\} \cup M | \begin{array}{l} m \in mv(m_1), \\ valid(m, M) \end{array} \right\})) & : \quad otherwise \end{cases} \tag{14.60}$$

Where function $valid(m, M)$ checks if the new moving candidate $m = (A'', B'')$ is valid. That neither A'' nor B'' is 3 or 6, and m hasn't been tried before in M to avoid any repeatedly attempts.

$$valid(m, M) = A'' \neq 3, A'' \neq 6, B'' \neq 3, B'' \neq 6, m \notin M \tag{14.61}$$

The following example Haskell program implements this solution. Note that it uses a plain list to represent the queue for illustration purpose.

```
import Data.Bits

solve = bfsSolve [[(15, 0)]] where
    bfsSolve :: [[(Int, Int)]] → [(Int, Int)]
    bfsSolve [] = [] -- no solution
    bfsSolve (c:cs) | (fst $ head c) == 0 = reverse c
                    | otherwise = bfsSolve (cs ++ map (:c)
                                        (filter (`valid` c) $ moves $ head c))
    valid (a, b) r = not $ or [ a `elem` [3, 6], b `elem` [3, 6],
                                (a, b) `elem` r]

moves (a, b) = if b < 8 then trans a b else map swap (trans b a) where
    trans x y = [(x - 8 - i, y + 8 + i)
                    | i ←[0, 1, 2, 4], i == 0 || (x ∘ &. i) /= 0]
    swap (x, y) = (y, x)
```

This algorithm can be easily modified to find all the possible solutions, but not just stop after finding the first one. This is left as the exercise to the reader. The following shows the two best solutions to this puzzle.

Solution 1:

Left	river	Right
wolf, goat, cabbage, farmer		
wolf, cabbage		goat, farmer
wolf, cabbage, farmer		goat
cabbage		wolf, goat, farmer
goat, cabbage, farmer		wolf
goat		wolf, cabbage, farmer
goat, farmer		wolf, cabbage
		wolf, goat, cabbage, farmer

Solution 2:

Left	river	Right
wolf, goat, cabbage, farmer		
wolf, cabbage		goat, farmer
wolf, cabbage, farmer		goat
wolf		goat, cabbage, farmer
wolf, goat, farmer		cabbage
goat		wolf, cabbage, farmer
goat, farmer		wolf, cabbage
		wolf, goat, cabbage, farmer

This algorithm can also be realized imperatively. Observing that our solution is in tail recursive manner, we can translate it directly to a loop. We use a list S to hold all the solutions can be found. The singleton list $\{(15, 0)\}$ is pushed to queue when initializing. As long as the queue isn't empty, we extract the head C from the queue by calling DeQ procedure. Examine if it reaches the final goal, if not, we expand all the possible moves and push to the tail of the queue for further searching.

1: **function** Solve
2: $S \leftarrow \Phi$
3: $Q \leftarrow \Phi$
4: EnQ$(Q, \{(15, 0)\})$
5: **while** $Q \neq \Phi$ **do**
6: $C \leftarrow$ DeQ(Q)

```
7:        if c_1 = (0, 15) then
8:            ADD(S, REVERSE(C))
9:        else
10:           for ∀m ∈ MOVES(C) do
11:               if VALID(m, C) then
12:                   ENQ(Q, {m} ∪ C)
13:    return S
```

Where MOVES, and VALID procedures are as same as before. The following Python example program implements this imperative algorithm.

```
def solve():
    s = []
    queue = [[(0xf, 0)]]
    while queue != []:
        cur = queue.pop(0)
        if cur[0] == (0, 0xf):
            s.append(reverse(cur))
        else:
            for m in moves(cur):
                queue.append([m]+cur)
    return s

def moves(s):
    (a, b) = s[0]
    return valid(s, trans(a, b) if b < 8 else swaps(trans(b, a)))

def valid(s, mv):
    return [(a, b) for (a, b) in mv
        if a not in [3, 6] and b not in [3, 6] and (a, b) not in s]

def trans(a, b):
    masks = [ 8 | (1<<i) for i in range(4)]
    return [(a ^ mask, b | mask) for mask in masks if a & mask == mask]

def swaps(s):
    return [(b, a) for (a, b) in s]
```

There is a minor difference between the program and the pseudo code, that the function to generate candidate moving options filters the invalid cases inside it.

Every time, no matter the farmer drives the boat back and forth, there are m options for him to choose, where m is the number of objects on the river bank the farmer drives from. m is always less than 4, that the algorithm won't take more than n^4 times at step n. This estimation is far more than the actual time, because we avoid trying all invalid cases. Our solution examines all the possible moving in the worst case. Because we check recorded steps to avoid repeated attempt, the algorithm takes about $O(n^2)$ time to search for n possible steps.

Water jugs puzzle

This is a popular puzzle in classic AI. The history of it should be very long. It says that there are two jugs, one is 9 quarts, the other is 4 quarts. How to use them to bring up from the river exactly 6 quarts of water?

There are varies versions of this puzzle, although the volume of the jugs, and the target volume of water differ. The solver is said to be young Blaise Pascal when he was a child, the French mathematician, scientist in one story, and Simèon Denis Poisson in another story. Later in the popular Hollywood movie 'Die-Hard 3', actor Bruce Willis and Samuel L. Jackson were also confronted with this puzzle.

Pòlya gave a nice way to solve this problem backwards in [14].

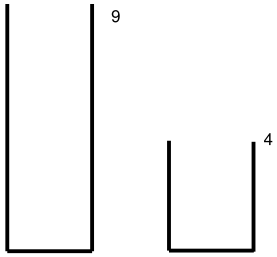


Figure 14.37: Two jugs with volume of 9 and 4.

Instead of thinking from the starting state as shown in figure 14.37. Pòlya pointed out that there will be 6 quarts of water in the bigger jugs at the final stage, which indicates the second last step, we can fill the 9 quarts jug, then pour out 3 quarts from it. In order to achieve this, there should be 1 quart of water left in the smaller jug as shown in figure 14.38.

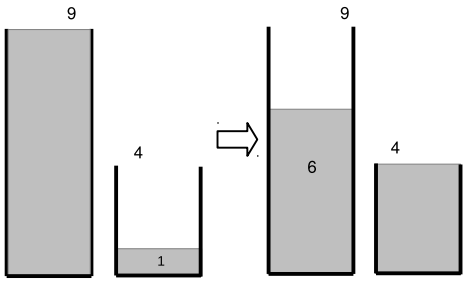


Figure 14.38: The last two steps.

It's easy to see that fill the 9 quarters jug, then pour to the 4 quarters jug twice can bring 1 quarters of water. As shown in figure 14.39. At this stage, we've found the solution. By reversing our findings, we can give the correct steps to bring exactly 6 quarters of water.

Pòlya's methodology is general. It's still hard to solve it without concrete algorithm. For instance, how to bring up 2 gallons of water from 899 and 1147 gallon jugs?

There are 6 ways to deal with 2 jugs in total. Denote the smaller jug as A, the bigger jug as B.

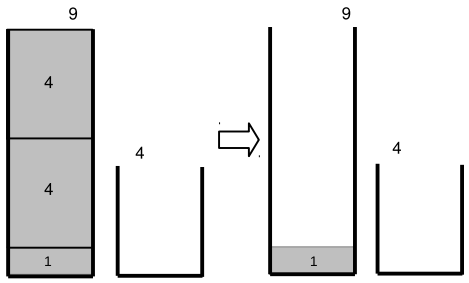


Figure 14.39: Fill the bigger jugs, and pour to the smaller one twice.

- Fill jug A from the river;
- Fill jug B from the river;
- Empty jug A;
- Empty jug B;
- Pour water from jug A to B;
- Pour water from jug B to A.

The following sequence shows an example. Note that in this example, we assume that $a < b < 2a$.

A	B	operation
0	0	start
a	0	fill A
0	a	pour A into B
a	a	fill A
2a - b	b	pour A into B
2a - b	0	empty B
0	2a - b	pour A into B
a	2a - b	fill A
3a - 2b	b	pour A into B
...	...	...

No matter what the above operations are taken, the amount of water in each jug can be expressed as $xa + yb$, where a and b are volumes of jugs, for some integers x and y. All the amounts of water we can get are linear combination of a and b. We can immediately tell given two jugs, if a goal g is solvable or not.

For instance, we can't bring 5 gallons of water with two jugs of volume 4 and 6 gallon. The number theory ensures that, the 2 water jugs puzzle can be solved if and only if g can be divided by the greatest common divisor of a and b. Written as:

$$gcd(a, b)|g \tag{14.62}$$

Where $m|n$ means n can be divided by m. What's more, if a and b are relatively prime, which means $gcd(a, b) = 1$, it's possible to bring up any quantity g of water.

Although $gcd(a, b)$ enables us to determine if the puzzle is solvable, it doesn't give us the detailed pour sequence. If we can find some integer x and y, so that $g = xa + yb$. We can arrange a sequence of operations (even it may not be the best solution) to solve it. The idea is that, without loss of generality, suppose $x > 0, y < 0$, we need fill jug A by x times, and empty jug B by y times in total.

Let's take $a = 3$, $b = 5$, and $g = 4$ for example, since $4 = 3 \times 3 - 5$, we can arrange a sequence like the following.

A	B	operation
0	0	start
3	0	fill A
0	3	pour A into B
3	3	fill A
1	5	pour A into B
1	0	empty B
0	1	pour A into B
3	1	fill A
0	4	pour A into B

In this sequence, we fill A by 3 times, and empty B by 1 time. The procedure can be described as the following:

Repeat x times:

1. Fill jug A;
2. Pour jug A into jug B, whenever B is full, empty it.

So the only problem left is to find the x and y. There is a powerful tool in number theory called, *Extended Euclid algorithm*, which can achieve this. Compare to the classic Euclid GCD algorithm, which can only give the greatest common divisor, The extended Euclid algorithm can give a pair of x, y as well, so that:

$$(d, x, y) = gcd_{ext}(a, b) \tag{14.63}$$

where $d = gcd(a, b)$ and $ax + by = d$. Without loss of generality, suppose $a < b$, there exits quotation q and remainder r that:

$$b = aq + r \tag{14.64}$$

Since d is the common divisor, it can divide both a and b, thus d can divide r as well. Because r is less than a, we can scale down the problem by finding GCD of a and r:

$$(d, x', y') = gcd_{ext}(r, a) \tag{14.65}$$

Where $d = x'r + y'a$ according to the definition of the extended Euclid algorithm. Transform $b = aq + r$ to $r = b - aq$, substitute r in above equation yields:

$$\begin{aligned} d &= x'(b - aq) + y'a \\ &= (y' - x'q)a + x'b \end{aligned} \tag{14.66}$$

This is the linear combination of a and b, so that we have:

$$\begin{cases} x = y' - x'\dfrac{b}{a} \\ y = x' \end{cases} \tag{14.67}$$

Note that this is a typical recursive relationship. The edge case happens when $a = 0$.

$$gcd(0, b) = b = 0a + 1b \tag{14.68}$$

Summarize the above result, the extended Euclid algorithm can be defined as the following:

$$gcd_{ext}(a, b) = \begin{cases} (b, 0, 1) & : \quad a = 0 \\ (d, y' - x'\dfrac{b}{a}, x') & : \quad otherwise \end{cases} \tag{14.69}$$

Where d, x', y' are defined in equation (14.65).

The 2 water jugs puzzle is almost solved, but there are still two detailed problems need to be tackled. First, extended Euclid algorithm gives the linear combination for the greatest common divisor d. While the target volume of water g isn't necessarily equal to d. This can be easily solved by multiplying x and y by m times, where $m = g/gcd(a, b)$; Second, we assume $x > 0$, to form a procedure to fill jug A with x times. However, the extended Euclid algorithm doesn't ensure x to be positive. For instance $gcd_{ext}(4, 9) = (1, -2, 1)$. Whenever we get a negative x, since $d = xa + yb$, we can continuously add b to x, and decrease y by a till a is greater than zero.

At this stage, we are able to give the complete solution to the 2 water jugs puzzle. Below is an example Haskell program.

```
extGcd 0 b = (b, 0, 1)
extGcd a b = let (d, x', y') = extGcd (b `mod` a) a in
               (d, y' - x' * (b `div` a), x')

solve a b g | g `mod` d /= 0 = [] -- no solution
            | otherwise = solve' (x * g `div` d)
    where
      (d, x, y) = extGcd a b
      solve' x | x < 0 = solve' (x + b)
               | otherwise = pour x [(0, 0)]
      pour 0 ps = reverse ((0, g):ps)
      pour x ps@((a', b'):_) | a' == 0 = pour (x - 1) ((a, b'):ps) -- fill a
                             | b' == b = pour x ((a', 0):ps) -- empty b
                             | otherwise = pour x ((max 0 (a' + b' - b),
                                                    min (a' + b') b):ps)
```

Although we can solve the 2 water jugs puzzle with extended Euclid algorithm, the solution may not be the best. For instance, when we are going to bring 4 gallons of water from 3 and 5 gallons jugs. The extended Euclid algorithm produces the following sequence:

```
[(0,0),(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),
(0,4),(3,4),(2,5),(2,0),(0,2),(3,2),(0,5),(3,5),
(3,0),(0,3),(3,3),(1,5),(1,0),(0,1),(3,1),(0,4)]
```

It takes 23 steps to achieve the goal, while the best solution only need 6 steps:

```
[(0,0),(0,5),(3,2),(0,2),(2,0),(2,5),(3,4)]
```

Observe the 23 steps, and we can find that jug B has already contained 4 gallons of water at the 8-th step. But the algorithm ignores this fact and goes on executing the left 15 steps. The reason is that the linear combination x and y we find with the extended Euclid algorithm are not the only numbers satisfying $g = xa + by$. For all these numbers, the smaller $|x| + |y|$, the less steps are needed. There is an exercise to addressing this problem in this section.

The interesting problem is how to find the best solution? We have two approaches, one is to find x and y to minimize $|x| + |y|$; the other is to adopt the quite similar idea as the wolf-goat-cabbage puzzle. We focus on the latter in this section. Since there are at most 6 possible options: fill A, fill B, pour A into B, pour B into A, empty A and empty B, we can try them in parallel, and check which decision can lead to the best solution. We need record all the states we've achieved to avoid any potential repetition. In order to realize this parallel approach with reasonable resources, a queue can be used to arrange our attempts. The elements stored in this queue are series of pairs (p, q), where p and q represent the volume of waters contained in each jug. These pairs record the sequence of our operations from the beginning to the latest. We initialize the queue with the singleton list contains the starting state $\{(0, 0)\}$.

$$solve(a, b, g) = solve'\{\{(0, 0)\}\} \tag{14.70}$$

Every time, when the queue isn't empty, we pick a sequence from the head of the queue. If this sequence ends with a pair contains the target volume g, we find a solution, we can print this sequence by reversing it; Otherwise, we expand the latest pair by trying all the possible 6 options, remove any duplicated states, and add them to the tail of the queue. Denote the queue as Q, the first sequence stored on the head of the queue as S, the latest pair in S as (p, q), and the rest of pairs as S'. After popping the head element, the queue becomes Q'. This algorithm can be defined like below:

$$solve'(Q) = \begin{cases} \Phi & : & Q = \Phi \\ reverse(S) & : & p = g \vee q = g \\ solve'(EnQ'(Q', \{\{s'\} \cup S' | s' \in try(S)\})) & : & otherwise \end{cases} \tag{14.71}$$

Where function EnQ' pushes a list of sequence to the queue one by one. Function $try(S)$ will try all possible 6 options to generate new pairs of water volumes:

$$try(S) = \{s' | s' \in \begin{Bmatrix} fillA(p, q), fillB(p, q), \\ pourA(p, q), pourB(p, q), \\ emptyA(p, q), emptyB(p, q) \end{Bmatrix}, s' \notin S'\} \tag{14.72}$$

It's intuitive to define the 6 options. For fill operations, the result is that the volume of the filled jug is full; for empty operation, the result volume is empty;

for pour operation, we need test if the jug is big enough to hold all the water.

$$\begin{array}{ll} fillA(p,q) = (a,q) & fillB(p,q) = (p,b) \\ emptyA(p,q) = (0,q) & emptyB(p,q) = (p,0) \\ pourA(p,q) = (max(0, p+q-b), min(x+y, b)) & \\ pourB(p,q) = (min(x+y, a), max(0, x+y-a)) & \end{array} \tag{14.73}$$

The following example Haskell program implements this method:

```
solve' a b g = bfs [[(0, 0)]] where
  bfs [] = []
  bfs (c:cs) | fst (head c) == g || snd (head c) == g = reverse c
             | otherwise = bfs (cs ++ map (:c) (expand c))
  expand ((x, y):ps) = filter (`notElem` ps) $ map (λf → f x y)
                              [fillA, fillB, pourA, pourB, emptyA, emptyB]
  fillA _ y = (a, y)
  fillB x _ = (x, b)
  emptyA _ y = (0, y)
  emptyB x _ = (x, 0)
  pourA x y = (max 0 (x + y - b), min (x + y) b)
  pourB x y = (min (x + y) a, max 0 (x + y - a))
```

This method always returns the fast solution. It can also be realized in imperative approach. Instead of storing the complete sequence of operations in every element in the queue, we can store the unique state in a global history list, and use links to track the operation sequence, this can save spaces.

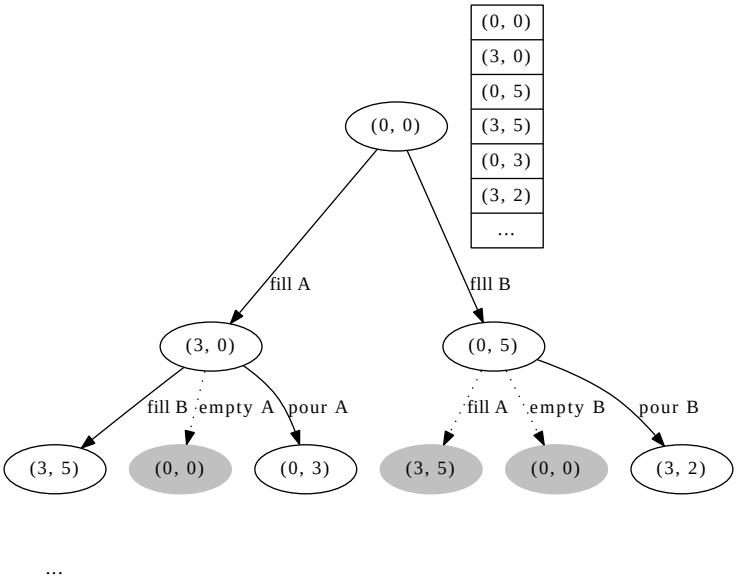


Figure 14.40: All attempted states are stored in a global list.

The idea is illustrated in figure 14.40. The initial state is (0, 0). Only 'fill A' and 'fill B' are possible. They are tried and added to the record list; Next we can try and record 'fill B' on top of (3, 0), which yields new state (3, 5).

However, when try 'empty A' from state (3, 0), we would return to the start state (0, 0). As this previous state has been recorded, it is ignored. All the repeated states are in gray color in this figure.

With such settings, we needn't remember the operation sequence in each element in the queue explicitly. We can add a 'parent' link to each node in figure 14.40, and use it to back-traverse to the starting point from any state. The following example ANSI C code shows such a definition.

```
struct Step {
    int p, q;
    struct Step* parent;
};

struct Step* make_step(int p, int q, struct Step* parent) {
    struct Step* s = (struct Step*) malloc(sizeof(struct Step));
    s→p = p;
    s→q = q;
    s→parent = parent;
    return s;
}
```

Where p, q are volumes of water in the 2 jugs. For any state s, define functions $p(s)$ and $q(s)$ return these 2 values, the imperative algorithm can be realized based on this idea as below.

```
1: function SOLVE(a, b, g)
2:     Q ← Φ
3:     PUSH-AND-RECORD(Q, (0, 0))
4:     while Q ≠ Φ do
5:         s ← POP(Q)
6:         if p(s) = g ∨ q(s) = g then
7:             return s
8:         else
9:             C ← EXPAND(s)
10:            for ∀c ∈ C do
11:                if c ≠ s ∧ ¬ VISITED(c) then
12:                    PUSH-AND-RECORD(Q, c)
13:    return NIL
```

Where PUSH-AND-RECORD does not only push an element to the queue, but also record this element as visited, so that we can check if an element has been visited before in the future. This can be implemented with a list. All push operations append the new elements to the tail. For pop operation, instead of removing the element pointed by head, the head pointer only advances to the next one. This list contains historic data which has to be reset explicitly. The following ANSI C code illustrates this idea.

```
struct Step *steps[1000], **head, **tail = steps;

void push(struct Step* s) { *tail++ = s; }

struct Step* pop() { return *head++; }

int empty() { return head == tail; }
```

```
void reset() {
    struct Step **p;
    for (p = steps; p != tail; ++p)
        free(*p);
    head = tail = steps;
}
```

In order to test a state has been visited, we can traverse the list to compare p and q.

```
int eq(struct Step* a, struct Step* b) {
    return a->p == b->p && a->q == b->q;
}

int visited(struct Step* s) {
    struct Step **p;
    for (p = steps; p != tail; ++p)
        if (eq(*p, s)) return 1;
    return 0;
}
```

The main program can be implemented as below:

```
struct Step* solve(int a, int b, int g) {
    int i;
    struct Step *cur, *cs[6];
    reset();
    push(make_step(0, 0, NULL));
    while (!empty()) {
        cur = pop();
        if (cur->p == g || cur->q == g)
            return cur;
        else {
            expand(cur, a, b, cs);
            for (i = 0; i < 6; ++i)
                if(!eq(cur, cs[i]) && !visited(cs[i]))
                    push(cs[i]);
        }
    }
    return NULL;
}
```

Where function `expand` tries all the 6 possible options:

```
void expand(struct Step* s, int a, int b, struct Step** cs) {
    int p = s->p, q = s->q;
    cs[0] = make_step(a, q, s); /*fill A*/
    cs[1] = make_step(p, b, s); /*fill B*/
    cs[2] = make_step(0, q, s); /*empty A*/
    cs[3] = make_step(p, 0, s); /*empty B*/
    cs[4] = make_step(max(0, p + q - b), min(p + q, b), s); /*pour A*/
    cs[5] = make_step(min(p + q, a), max(0, p + q - a), s); /*pour B*/
}
```

And the result steps is back tracked in reversed order, it can be output with a recursive function:

```
void print(struct Step* s) {
    if (s) {
        print(s→parent);
        printf("%d, %d\n", s→p, s→q);
    }
}
```

Kloski

Kloski is a block sliding puzzle. It appears in many countries. There are different sizes and layouts. Figure 14.41 illustrates a traditional Kloski game in China.

(a) Initial layout of blocks (b) Block layout after several movements

Figure 14.41: 'Huarong Dao', the traditional Kloski game in China.

In this puzzle, there are 10 blocks, each is labeled with text or icon. The smallest block has size of 1 unit square, the biggest one is 2×2 units size. Note there is a slot of 2 units wide at the middle-bottom of the board. The biggest block represents a king in ancient time, while the others are enemies. The goal is to move the biggest block to the slot, so that the king can escape. This game is named as 'Huarong Dao', or 'Huarong Escape' in China. Figure 14.42 shows the similar Kloski puzzle in Japan. The biggest block means daughter, while the others are her family members. This game is named as 'Daughter in the box' in Japan (Japanese name: *hakoiri musume*).

Figure 14.42: 'Daughter in the box', the Kloski game in Japan.

In this section, we want to find a solution, which can slide blocks from the initial state to the final state with the minimum movements.

The intuitive idea to model this puzzle is to use a 5×4 matrix representing the board. All pieces are labeled with a number. The following matrix M, for example, shows the initial state of the puzzle.

$$M = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix}$$

In this matrix, the cells of value i mean the i-th piece covers this cell. The special value 0 represents a free cell. By using sequence 1, 2, ... to identify pieces, a special layout can be further simplified as an array L. Each element is a list of cells covered by the piece indexed with this element. For example, $L[4] = \{(3,2),(3,3)\}$ means the 4-th piece covers cells at position $(3,2)$ and $(3,3)$, where (i,j) means the cell at row i and column j.

The starting layout can be written as the following Array.

$$\{\{(1,1),(2,1)\},\{(1,4),(2,4)\},\{(3,1),(4,1)\},\{(3,2),(3,3)\},\{(3,4),(4,4)\},$$
$$\{(5,1)\},\{(4,2)\},\{(4,3)\},\{(5,4)\},\{(1,2),(1,3),(2,2),(2,3)\}\}$$

When moving the Kloski blocks, we need examine all the 10 blocks, checking each block if it can move up, down, left and right. it seems that this approach would lead to a very huge amount of possibilities, because each step might have 10×4 options, there will be about 40^n cases in the n-th step.

Actually, there won't be so much options. For example, in the first step, there are only 4 valid moving: the 6-th piece moves right; the 7-th and 8-th move down; and the 9-th moves left.

All others are invalid moving. Figure 14.43 shows how to test if the moving is possible.

The left example illustrates sliding block labeled with 1 down. There are two cells covered by this block. The upper 1 moves to the cell previously occupied by this same block, which is also labeled with 1; The lower 1 moves to a free cell, which is labeled with 0;

The right example, on the other hand, illustrates invalid sliding. In this case, the upper cells could move to the cell occupied by the same block. However, the lower cell labeled with 1 can't move to the cell occupied by other block, which is labeled with 2.

In order to test the valid moving, we need examine all the cells a block will cover. If they are labeled with 0 or a number as same as this block, the moving is valid. Otherwise it conflicts with some other block. For a layout L, the corresponding matrix is M, suppose we want to move the k-th block with $(\Delta x, \Delta y)$, where $|\Delta x| \leq 1, |\Delta y| \leq 1$. The following equation tells if the moving is valid:

$$\begin{aligned} &valid(L,k,\Delta x,\Delta y): \\ &\quad \forall (i,j) \in L[k] \Rightarrow \quad i' = i + \Delta y, j' = j + \Delta x, \\ &\qquad\qquad\qquad\quad (1,1) \leq (i',j') \leq (5,4), M_{i'j'} \in \{k,0\} \end{aligned} \tag{14.74}$$

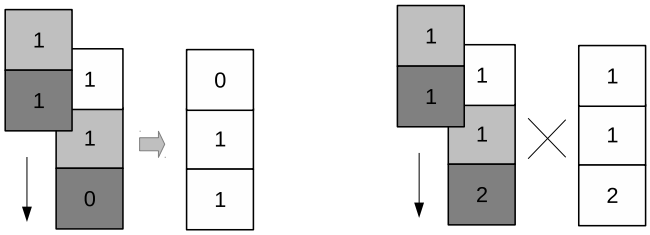


Figure 14.43: Left: both the upper and the lower 1 are OK; Right: the upper 1 is OK, the lower 1 conflicts with 2.

Another important point to solve Kloski puzzle, is about how to avoid repeated attempts. The obvious case is that after a series of sliding, we end up a matrix which have been transformed from. However, it is not enough to only avoid the same matrix. Consider the following two metrics. Although $M_1 \neq M_2$, we need drop options to M_2, because they are essentially the same.

$$M_1 = \begin{bmatrix} 1 & 10 & 10 & 2 \\ 1 & 10 & 10 & 2 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 8 & 5 \\ 6 & 0 & 0 & 9 \end{bmatrix} \quad M_2 = \begin{bmatrix} 2 & 10 & 10 & 1 \\ 2 & 10 & 10 & 1 \\ 3 & 4 & 4 & 5 \\ 3 & 7 & 6 & 5 \\ 8 & 0 & 0 & 9 \end{bmatrix}$$

This fact tells us, that we should compare the layout, but not merely matrix to avoid repetition. Denote the corresponding layouts as L_1 and L_2 respectively, it's easy to verify that $||L_1|| = ||L_2||$, where $||L||$ is the normalized layout, which is defined as below:

$$||L|| = sort(\{sort(l_i)|\forall l_i \in L\}) \tag{14.75}$$

In other words, a normalized layout is ordered for all its elements, and every element is also ordered. The ordering can be defined as that $(a, b) \leq (c, d) \Leftrightarrow an + b \leq cn + d$, where n is the width of the matrix.

Observing that the Kloski board is symmetric, thus a layout can be mirrored from another one. Mirrored layout is also a kind of repeating, which should be avoided. The following M_1 and M_2 show such an example.

$$M_1 = \begin{bmatrix} 10 & 10 & 1 & 2 \\ 10 & 10 & 1 & 2 \\ 3 & 5 & 4 & 4 \\ 3 & 5 & 8 & 9 \\ 6 & 7 & 0 & 0 \end{bmatrix} \quad M_2 = \begin{bmatrix} 3 & 1 & 10 & 10 \\ 3 & 1 & 10 & 10 \\ 4 & 4 & 2 & 5 \\ 7 & 6 & 2 & 5 \\ 0 & 0 & 9 & 8 \end{bmatrix}$$

Note that, the normalized layouts are symmetric to each other. It's easy to get a mirrored layout like this:

$$mirror(L) = \{\{(i, n - j + 1) | \forall (i, j) \in l\} | \forall l \in L\} \tag{14.76}$$

We find that the matrix representation is useful in validating the moving, while the layout is handy to model the moving and avoid repeated attempt. We can use the similar approach to solve the Kloski puzzle. We need a queue, every element in the queue contains two parts: a series of moving and the latest layout led by the moving. Each moving is in form of $(k, (\Delta y, \Delta x))$, which means moving the k-th block, with Δy in row, and Δx in column in the board.

The queue contains the starting layout when initialized. Whenever this queue isn't empty, we pick the first one from the head, checking if the biggest block is on target, that $L[10] = \{(4, 2), (4, 3), (5, 2), (5, 3)\}$. If yes, then we are done; otherwise, we try to move every block with 4 options: left, right, up, and down, and store all the possible, unique new layout to the tail of the queue. During this searching, we need record all the normalized layouts we've ever found to avoid any duplication.

Denote the queue as Q, the historic layouts as H, the first layout on the head of the queue as L, its corresponding matrix as M. and the moving sequence to this layout as S. The algorithm can be defined as the following.

$$solve(Q, H) = \begin{cases} \Phi & : & Q = \Phi \\ reverse(S) & : & L[10] = \{(4, 2), (4, 3), (5, 2), (5, 3)\} \\ solve(Q', H') & : & otherwise \end{cases} \tag{14.77}$$

The first clause says that if the queue is empty, we've tried all the possibilities and can't find a solution; The second clause finds a solution, it returns the moving sequence in reversed order; These are two edge cases. Otherwise, the algorithm expands the current layout, puts all the valid new layouts to the tail of the queue to yield Q', and updates the normalized layouts to H'. Then it performs recursive searching.

In order to expand a layout to valid unique new layouts, we can define a function as below:

$$\begin{aligned} expand(L, H) = \{(k, (\Delta y, \Delta x)) | \quad & \forall k \in \{1, 2, ..., 10\}, \\ & \forall (\Delta y, \Delta x) \in \{(0, -1), (0, 1), (-1, 0), (1, 0)\}, \\ & valid(L, k, \Delta x, \Delta y), unique(L', H)\} \end{aligned} \tag{14.78}$$

Where L' is the the new layout by moving the k-th block with $(\Delta y, \Delta x)$ from L, M' is the corresponding matrix, and M'' is the matrix to the mirrored layout of L'. Function unique is defined like this:

$$unique(L', H) = M' \notin H \wedge M'' \notin H \tag{14.79}$$

We'll next show some example Haskell Kloski programs. As array isn't mutable in the purely functional settings, tree based map is used to represent layout [10]. Some type synonyms are defined as below:

```
import qualified Data.Map as M
import Data.Ix
import Data.List (sort)

type Point = (Integer, Integer)
type Layout = M.Map Integer [Point]
type Move = (Integer, Point)

data Ops = Op Layout [Move]
```

The main program is almost as same as the $sort(Q, H)$ function defined above.

```
solve :: [Ops] → [[[Point]]]→ [Move]
solve [] _ = [] -- no solution
solve (Op x seq : cs) visit
    | M.lookup 10 x == Just [(4, 2), (4, 3), (5, 2), (5, 3)] = reverse seq
    | otherwise = solve q visit'
  where
    ops = expand x visit
    visit' = map (layout ∘ move x) ops ++ visit
    q = cs ++ [Op (move x op) (op:seq) | op ← ops ]
```

Where function `layout` gives the normalized form by sorting. `move` returns the updated map by sliding the i-th block with $(\Delta y, \Delta x)$.

```
layout = sort ∘ map sort ∘ M.elems

move x (i, d) = M.update (Just ∘ map (flip shift d)) i x

shift (y, x) (dy, dx) = (y + dy, x + dx)
```

Function `expand` gives all the possible new options. It can be directly translated from $expand(L, H)$.

```
expand :: Layout → [[[Point]]] → [Move]
expand x visit = [(i, d) | i ←[1..10],
                           d ← [(0, -1), (0, 1), (-1, 0), (1, 0)],
                           valid i d, unique i d] where
  valid i d = all (λp → let p' = shift p d in
                    inRange (bounds board) p' &&
                    (M.keys $ M.filter (elem p') x) `elem` [[i], []])
              (maybe [] id $ M.lookup i x)
  unique i d = let mv = move x (i, d) in
               all (`notElem` visit) (map layout [mv, mirror mv])
```

Note that we also filter out the mirrored layouts. The `mirror` function is given as the following.

```
mirror = M.map (map (λ (y, x) → (y, 5 - x)))
```

This program takes several minutes to produce the best solution, which takes 116 steps. The final 3 steps are shown as below:

[10] Alternatively, finger tree based sequence shown in previous chapter can be used

```
...

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['A', 'A', '6', '0']
['A', 'A', '0', '8']

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['A', 'A', '0', '6']
['A', 'A', '0', '8']

['5', '3', '2', '1']
['5', '3', '2', '1']
['7', '9', '4', '4']
['0', 'A', 'A', '6']
['0', 'A', 'A', '8']

total 116 steps
```

The Kloski solution can also be realized imperatively. Note that the $solve(Q, H)$ is tail-recursive, it's easy to transform the algorithm with looping. We can also link one layout to its parent, so that the moving sequence can be recorded globally. This can save some spaces, as the queue needn't store the moving information in every element. When output the result, we only need back-tracking to the starting layout from the last one.

Suppose function LINK(L', L) links a new layout L' to its parent layout L. The following algorithm takes a starting layout, and searches for best moving sequence.

```
1: function SOLVE(L_0)
2:     H ← ||L_0||
3:     Q ← Φ
4:     PUSH(Q, LINK(L_0, NIL))
5:     while Q ≠ Φ do
6:         L ← POP(Q)
7:         if L[10] = {(4,2),(4,3),(5,2),(5,3)} then
8:             return L
9:         else
10:            for each L' ∈ EXPAND(L, H) do
11:                PUSH(Q, LINK(L', L))
12:                APPEND(H, ||L'||)
13:    return NIL                                  ▷ No solution
```

The following example Python program implements this algorithm:

```
class Node:
    def __init__(self, l, p = None):
        self.layout = l
        self.parent = p
```

```
def solve(start):
    visit = [normalize(start)]
    queue = [Node(start)]
    while queue != []:
        cur = queue.pop(0)
        layout = cur.layout
        if layout[-1] == [(4, 2), (4, 3), (5, 2), (5, 3)]:
            return cur
        else:
            for brd in expand(layout, visit):
                queue.append(Node(brd, cur))
                visit.append(normalize(brd))
    return None # no solution
```

Where `normalize` and `expand` are implemented as below:

```
def normalize(layout):
    return sorted([sorted(r) for r in layout])

def expand(layout, visit):
    def bound(y, x):
        return 1 ≤ y and y ≤ 5 and 1 ≤ x and x ≤ 4
    def valid(m, i, y, x):
        return m[y - 1][x - 1] in [0, i]
    def unique(brd):
        (m, n) = (normalize(brd), normalize(mirror(brd)))
        return all(m != v and n != v for v in visit)
    s = []
    d = [(0, -1), (0, 1), (-1, 0), (1, 0)]
    m = matrix(layout)
    for i in range(1, 11):
        for (dy, dx) in d:
            if all(bound(y + dy, x + dx) and valid(m, i, y + dy, x + dx)
                    for (y, x) in layout[i - 1]):
                brd = move(layout, (i, (dy, dx)))
                if unique(brd):
                    s.append(brd)
    return s
```

Like most programming languages, arrays are indexed from 0 but not 1 in Python. This has to be handled properly. The rest functions including `mirror`, `matrix`, and `move` are implemented as the following.

```
def mirror(layout):
    return [[(y, 5 - x) for (y, x) in r] for r in layout]

def matrix(layout):
    m = [[0]*4 for _ in range(5)]
    for (i, ps) in zip(range(1, 11), layout):
        for (y, x) in ps:
            m[y - 1][x - 1] = i
    return m

def move(layout, delta):
    (i, (dy, dx)) = delta
```

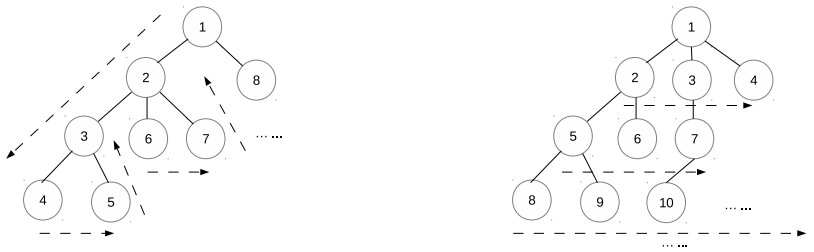
```
    m = dup(layout)
    m[i - 1] = [(y + dy, x + dx) for (y, x) in m[i - 1]]
    return m

def dup(layout):
    return [r[:] for r in layout]
```

It's possible to modify this Kloski algorithm, so that it does not only stop at the fast solution, but also search all the solutions. In such case, the computation time is bound to the size of a space V, where V holds all the layouts can be transformed from the starting layout. If all these layouts are stored globally, with a parent field point to the predecessor, the space requirement of this algorithm is also bound to $O(V)$.

Summary of BFS

The above three puzzles, the wolf-goat-cabbage puzzle, the water jugs puzzle, and the Kloski puzzle show some common solution structure. Similar to the DFS problems, they all have the starting state and the end state. The wolf-goat-cabbage puzzle starts with the wolf, the goat, the cabbage and the farmer all in one side, while the other side is empty. It ends up in a state that they all moved to the other side. The water jugs puzzle starts with two empty jugs, and ends with either jug contains a certain volume of water. The Kloski puzzle starts from a layout and ends to another layout that the biggest block begging slided to a given position.

All problems specify a set of rules which can transfer from one state to another. Different form the DFS approach, we try all the possible options 'in parallel'. We won't search further until all the other alternatives in the same step have been examined. This method ensures that the solution with the minimum steps can be found before those with more steps. Review and compare the two figures we've drawn before shows the difference between these two approaches. For the later one, because we expand the searching horizontally, it is called as Breadth-first search (BFS for short).



(a) Depth First Search (b) Breadth First Search

Figure 14.44: Search orders for DFS and BFS.

As we can't perform search really in parallel, BFS realization typically utilizes a queue to store the search options. The candidate with less steps pops

from the head, while the new candidate with more steps is pushed to the tail of the queue. Note that the queue should meet constant time enqueue and dequeue requirement, which we've explained in previous chapter of queue. Strictly speaking, the example functional programs shown above don't meet this criteria. They use list to mimic queue, which can only provide linear time pushing. Readers can replace them with the functional queue we explained before.

BFS provides a simple method to search for optimal solutions in terms of the number of steps. However, it can't search for more general optimal solution. Consider another directed graph as shown in figure 14.45, the length of each section varies. We can't use BFS to find the shortest route from one city to another.

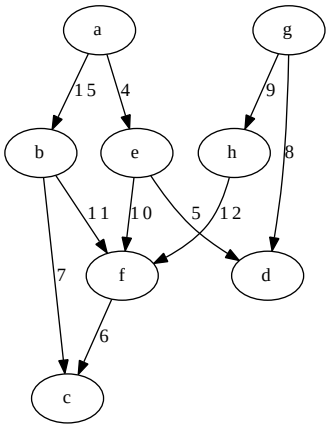


Figure 14.45: A weighted directed graph.

Note that the shortest route from city a to city c isn't the one with the fewest steps $a \to b \to c$. The total length of this route is 22; But the route with more steps $a \to e \to f \to c$ is the best. The length of it is 20. The coming sections introduce other algorithms to search for optimal solution.

14.3.2 Search the optimal solution

Searching for the optimal solution is quite important in many aspects. People need the 'best' solution to save time, space, cost, or energy. However, it's not easy to find the best solution with limited resources. There have been many optimal problems can only be solved by brute-force. Nevertheless, we've found that, for some of them, There exists special simplified ways to search the optimal solution.

Grady algorithm

Huffman coding

Huffman coding is a solution to encode information with the shortest length of code. Consider the popular ASCII code, which uses 7 bits to encode characters, digits, and symbols. ASCII code can represent $2^7 = 128$ different symbols. With 0, 1 bits, we need at least $\log_2 n$ bits to distinguish n different symbols. For text with only case insensitive English letters, we can define a code table like below.

char	code	char	code
A	00000	N	01101
B	00001	O	01110
C	00010	P	01111
D	00011	Q	10000
E	00100	R	10001
F	00101	S	10010
G	00110	T	10011
H	00111	U	10100
I	01000	V	10101
J	01001	W	10110
K	01010	X	10111
L	01011	Y	11000
M	01100	Z	11001

With this code table, text 'INTERNATIONAL' is encoded to 65 bits.

```
00010101101100100100100011011000000110010001001110101100000011010
```

Observe the above code table, which actually maps the letter 'A' to 'Z' from 0 to 25. There are 5 bits to represent every code. Code zero is forced as '00000' but not '0' for example. Such kind of coding method, is called fixed-length coding.

Another coding method is variable-length coding. That we can use just one bit '0' for 'A', two bits '10' for C, and 5 bits '11001' for 'Z'. Although this approach can shorten the total length of the code for 'INTERNATIONAL' from 65 bits dramatically, it causes problem when decoding. When processing a sequence of bits like '1101', we don't know if it means '1' followed by '101', which stands for 'BF'; or '110' followed by '1', which is 'GB', or '1101' which is 'N'.

The famous Morse code is variable-length coding system. That the most used letter 'E' is encoded as a dot, while 'Z' is encoded as two dashes and two dots. Morse code uses a special pause separator to indicate the termination of a code, so that the above problem won't happen. There is another solution to avoid ambiguity. Consider the following code table.

char	code	char	code
A	110	E	1110
I	101	L	1111
N	01	O	000
R	001	T	100

Text 'INTERNATIONAL' is encoded to 38 bits only:

```
10101100111000101110100101000011101111
```

If decode the bits against the above code table, we won't meet any ambiguity symbols. This is because there is no code for any symbol is the prefix of another one. Such code is called *prefix-code*. (You may wonder why it isn't called as non-prefix code.) By using prefix-code, we needn't separators at all. So that the length of the code can be shorten.

This is a very interesting problem. Can we find a prefix-code table, which produce the shortest code for a given text? The very same problem was given to David A. Huffman in 1951, who was still a student in MIT[15]. His professor

Robert M. Fano told the class that those who could solve this problem needn't take the final exam. Huffman almost gave up and started preparing the final exam when he found the most efficient answer.

The idea is to create the coding table according to the frequency of the symbol appeared in the text. The more used symbol is assigned with the shorter code.

It's not hard to process some text, and calculate the occurrence for each symbol. So that we have a symbol set, each one is augmented with a weight. The weight can be the number which indicates the frequency this symbol occurs. We can use the number of occurrence, or the probabilities for example.

Huffman discovered that a binary tree can be used to generate prefix-code. All symbols are stored in the leaf nodes. The codes are generated by traversing the tree from root. When go left, we add a zero; and when go right we add a one.

Figure 14.46 illustrates a binary tree. Taking symbol 'N' for example, starting from the root, we first go left, then right and arrive at 'N'. Thus the code for 'N' is '01'; While for symbol 'A', we can go right, right, then left. So 'A' is encode to '110'. Note that this approach ensures none code is the prefix of the other.

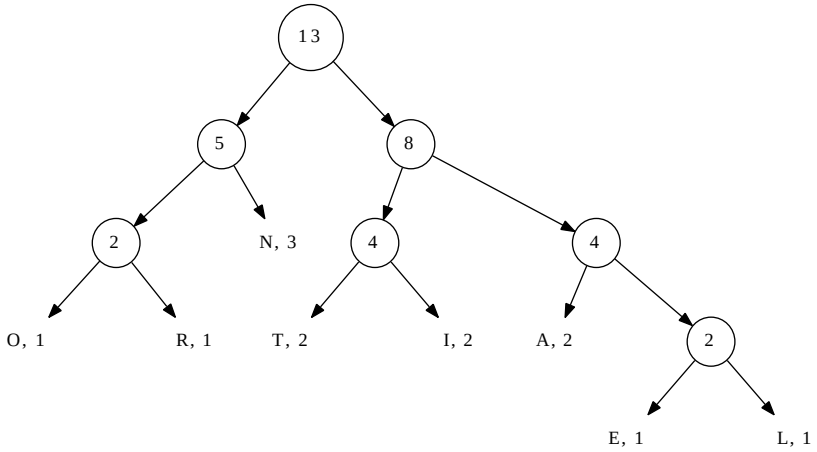
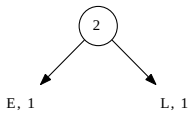


Figure 14.46: An encoding tree.

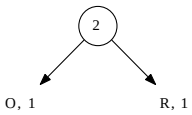
Note that this tree can also be used directly for decoding. When scan a series of bits, if the bit is zero, we go left; if the bit is one, we go right. When arrive at a leaf, we decode a symbol from that leaf. And we restart from the root of the tree for the coming bits.

Given a list of symbols with weights, we need build such a binary tree, so that the symbol with greater weight has shorter path from the root. Huffman developed a bottom-up solution. When start, all symbols are put into a leaf node. Every time, we pick two nodes, which has the smallest weight, and merge them into a branch node. The weight of this branch is the sum of its two children. We repeatedly pick the two smallest weighted nodes and merge till there is only one tree left. Figure 14.47 illustrates such a building process.

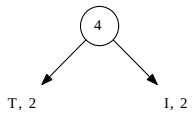
We can reuse the binary tree definition to formalize Huffman coding. We



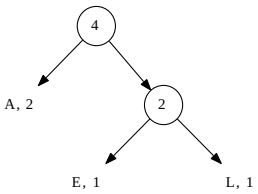
(a) 1.



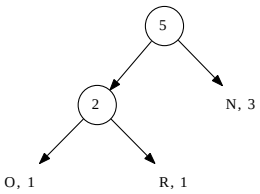
(b) 2.



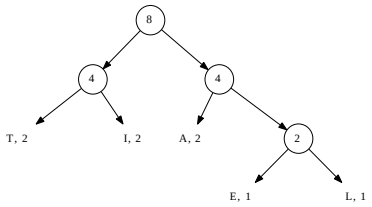
(c) 3.



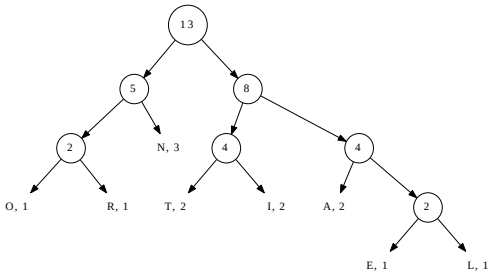
(d) 4.



(e) 5.



(f) 6.



(g) 7.

Figure 14.47: Steps to build a Huffman tree.

augment the weight information, and the symbols are only stored in leaf nodes. The following C like definition, shows an example.

```
struct Node {
    int w;
    char c;
    struct Node *left, *right;
};
```

Some limitation can be added to the definition, as empty tree isn't allowed. A Huffman tree is either a leaf, which contains a symbol and its weight; or a branch, which only holds total weight of all leaves. The following Haskell code, for instance, explicitly specifies these two cases.

```
data HTr w a = Leaf w a | Branch w (HTr w a) (HTr w a)
```

When merge two Huffman trees T_1 and T_2 to a bigger one, These two trees are set as its children. We can select either one as the left, and the other as the right. the weight of the result tree T is the sum of its two children. so that $w = w_1 + w_2$. Define $T_1 < T_2$ if $w_1 < w_2$, One possible Huffman tree building algorithm can be realized as the following.

$$build(A) = \begin{cases} T_1 & : \quad A = \{T_1\} \\ build(\{merge(T_a, T_b)\} \cup A') & : \quad otherwise \end{cases} \tag{14.80}$$

A is a list of trees. It is initialized as leaves for all symbols and their weights. If there is only one tree in this list, we are done, the tree is the final Huffman tree. Otherwise, The two smallest tree T_a and T_b are extracted, and the rest trees are hold in list A'. T_a and T_b are merged to one bigger tree, and put back to the tree list for further recursive building.

$$(T_a, T_b, A') = extract(A) \tag{14.81}$$

We can scan the tree list to extract the 2 nodes with the smallest weight. Below equation shows that when the scan begins, the first 2 elements are compared and initialized as the two minimum ones. An empty accumulator is passed as the last argument.

$$extract(A) = extract'(min(T_1, T_2), max(T_1, T_2), \{T_3, T_4, ...\}, \Phi) \tag{14.82}$$

For every tree, if its weight is less than the smallest two we've ever found, we update the result to contain this tree. For any given tree list A, denote the first tree in it as T_1, and the rest trees except T_1 as A'. The scan process can be defined as the following.

$$extract'(T_a, T_b, A, B) = \begin{cases} (T_a, T_b, B) & : \quad A = \Phi \\ extract'(T_a', T_b', A', \{T_b\} \cup A) & : \quad T_1 < T_b \\ extract'(T_a, T_b, A', \{T_1\} \cup A) & : \quad otherwise \end{cases} \tag{14.83}$$

Where $T_a' = min(T_1, T_a)$, $T_b' = max(T_1, T_a)$ are the updated two trees with the smallest weights.

The following Haskell example program implements this Huffman tree building algorithm.

```
build [x] = x
build xs = build ((merge x y) : xs') where
  (x, y, xs') = extract xs

extract (x:y:xs) = min2 (min x y) (max x y) xs [] where
  min2 x y [] xs = (x, y, xs)
  min2 x y (z:zs) xs | z < y = min2 (min z x) (max z x) zs (y:xs)
                     | otherwise = min2 x y zs (z:xs)
```

This building solution can also be realized imperatively. Given an array of Huffman nodes, we can use the last two cells to hold the nodes with the smallest weights. Then we scan the rest of the array from right to left. Whenever there is a node with the smaller weight, this node will be exchanged with the bigger one of the last two. After all nodes have been examined, we merge the trees in the last two cells, and drop the last cell. This shrinks the array by one. We repeat this process till there is only one tree left.

1: **function** HUFFMAN(A)
2: **while** $|A| > 1$ **do**
3: $n \leftarrow |A|$
4: **for** $i \leftarrow n - 2$ down to 1 **do**
5: **if** $A[i] <$ MAX$(A[n], A[n-1])$ **then**
6: EXCHANGE $A[i] \leftrightarrow$ MAX$(A[n], A[n-1])$
7: $A[n-1] \leftarrow$ MERGE$(A[n], A[n-1])$
8: DROP$(A[n])$
9: **return** $A[1]$

The following C++ example program implements this algorithm. Note that this algorithm needn't the last two elements being ordered.

```
typedef vector<Node*> Nodes;

bool lessp(Node* a, Node* b) { return a->w < b->w; }

Node* max(Node* a, Node* b) { return lessp(a, b) ? b : a; }

void swap(Nodes& ts, int i, int j, int k) {
    swap(ts[i], ts[ts[j] < ts[k] ? k : j]);
}

Node* huffman(Nodes ts) {
    int n;
    while((n = ts.size()) > 1) {
        for (int i = n - 3; i >= 0; --i)
            if (lessp(ts[i], max(ts[n-1], ts[n-2])))
                swap(ts, i, n-1, n-2);
        ts[n-2] = merge(ts[n-1], ts[n-2]);
        ts.pop_back();
    }
    return ts.front();
}
```

The algorithm merges all the leaves, and it need scan the list in each iteration. Thus the performance is quadratic. This algorithm can be improved. Observe that each time, only the two trees with the smallest weights are merged. This

reminds us the heap data structure. Heap ensures to access the smallest element fast. We can put all the leaves in a heap. For binary heap, this is typically a linear operation. Then we extract the minimum element twice, merge them, then put the bigger tree back to the heap. This is $O(\lg n)$ operation if binary heap is used. So the total performance is $O(n \lg n)$, which is better than the above algorithm. The next algorithm extracts the node from the heap, and starts Huffman tree building.

$$build(H) = reduce(top(H), pop(H)) \tag{14.84}$$

This algorithm stops when the heap is empty; Otherwise, it extracts another nodes from the heap for merging.

$$reduce(T, H) = \begin{cases} T & : \quad H = \Phi \\ build(insert(merge(T, top(H)), pop(H))) & : \quad otherwise \end{cases} \tag{14.85}$$

Function *build* and *reduce* are mutually recursive. The following Haskell example program implements this algorithm by using heap defined in previous chapter.

```
huffman' :: (Num a, Ord a) ⇒ [(b, a)] → HTr a b
huffman' = build' ∘ Heap.fromList ∘ map (λ(c, w) → Leaf w c) where
  build' h = reduce (Heap.findMin h) (Heap.deleteMin h)
  reduce x Heap.E = x
  reduce x h = build' $ Heap.insert (Heap.deleteMin h) (merge x (Heap.findMin h))
```

The heap solution can also be realized imperatively. The leaves are firstly transformed to a heap, so that the one with the minimum weight is put on the top. As far as there are more than 1 elements in the heap, we extract the two smallest, merge them to a bigger one, and put back to the heap. The final tree left in the heap is the result Huffman tree.

1: **function** HUFFMAN'(A)
2: BUILD-HEAP(A)
3: **while** $|A| > 1$ **do**
4: $T_a \leftarrow$ HEAP-POP(A)
5: $T_b \leftarrow$ HEAP-POP(B)
6: HEAP-PUSH(A, MERGE(T_a, T_b))
7: **return** HEAP-POP(A)

The following example C++ code implements this heap solution. The heap used here is provided in the standard library. Because the max-heap, but not min-heap would be made by default, a greater predication is explicitly passed as argument.

```
bool greaterp(Node* a, Node* b) { return b→w < a→w; }

Node* pop(Nodes& h) {
    Node* m = h.front();
    pop_heap(h.begin(), h.end(), greaterp);
    h.pop_back();
    return m;
}
```

```
void push(Node* t, Nodes& h) {
    h.push_back(t);
    push_heap(h.begin(), h.end(), greaterp);
}

Node* huffman1(Nodes ts) {
    make_heap(ts.begin(), ts.end(), greaterp);
    while (ts.size() > 1) {
        Node* t1 = pop(ts);
        Node* t2 = pop(ts);
        push(merge(t1, t2), ts);
    }
    return ts.front();
}
```

When the symbol-weight list has been already sorted, there exists a linear time method to build the Huffman tree. Observe that during the Huffman tree building, it produces a series of merged trees with weight in ascending order. We can use a queue to manage the merged trees. Every time, we pick the two trees with the smallest weight from both the queue and the list, merge them and push the result to the queue. All the trees in the list will be processed, and there will be only one tree left in the queue. This tree is the result Huffman tree. This process starts by passing an empty queue as below.

$$build'(A) = reduce'(extract''(\Phi, A)) \tag{14.86}$$

Suppose A is in ascending order by weight, At any time, the tree with the smallest weight is either the header of the queue, or the first element of the list. Denote the header of the queue is T_a, after pops it, the queue is Q'; The first element in A is T_b, the rest elements are hold in A'. Function $extract''$ can be defined like the following.

$$extract''(Q, A) = \begin{cases} (T_b, (Q, A')) & : & Q = \Phi \\ (T_a, (Q', A)) & : & A = \Phi \vee T_a < T_b \\ (T_b, (Q, A')) & : & otherwise \end{cases} \tag{14.87}$$

Actually, the pair of queue and tree list can be viewed as a special heap. The tree with the minimum weight is continuously extracted and merged.

$$reduce'(T, (Q, A)) = \begin{cases} T & : & Q = \Phi \wedge A = \Phi \\ reduce'(extract''(push(Q'', merge(T, T')), A'')) & : & otherwise \end{cases} \tag{14.88}$$

Where $(T', (Q'', A'')) = extract''(Q, A)$, which means extracting another tree. The following Haskell example program shows the implementation of this method. Note that this program explicitly sort the leaves, which isn't necessary if the leaves are ordered. Again, the list, but not a real queue is used here for illustration purpose. List isn't good at pushing new element, please refer to the chapter of queue for details about it.

```
huffman'' :: (Num a, Ord a) ⇒ [(b, a)] → HTr a b
huffman'' = reduce ∘ wrap ∘ sort ∘ map (λ(c, w) → Leaf w c) where
```

```
wrap xs = delMin ([], xs)
reduce (x, ([], [])) = x
reduce (x, h) = let (y, (q, xs)) = delMin h in
                reduce $ delMin (q ++ [merge x y], xs)
delMin ([], (x:xs)) = (x, ([], xs))
delMin ((q:qs), []) = (q, (qs, []))
delMin ((q:qs), (x:xs)) | q < x = (q, (qs, (x:xs)))
                        | otherwise = (x, ((q:qs), xs))
```

This algorithm can also be realized imperatively.

```
1: function HUFFMAN"(A)                          ▷ A is ordered by weight
2:     Q ← Φ
3:     T ← EXTRACT(Q, A)
4:     while Q ≠ Φ ∨ A ≠ Φ do
5:         PUSH(Q, MERGE(T, EXTRACT(Q, A)))
6:         T ← EXTRACT(Q, A)
7:     return T
```

Where function EXTRACT(Q, A) extracts the tree with the smallest weight from the queue and the list. It mutates the queue and tree if necessary. Denote the head of the queue is T_a, and the first element of the list as T_b.

```
1: function EXTRACT(Q, A)
2:     if Q ≠ Φ ∧ (A = Φ ∨ T_a < T_b) then
3:         return POP(Q)
4:     else
5:         return DETACH(A)
```

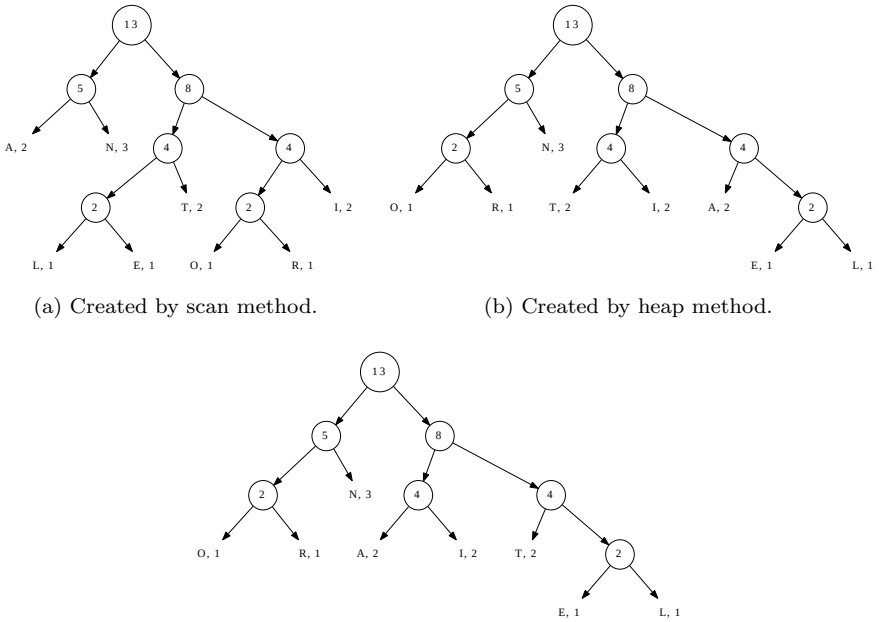
Where procedure DETACH(A), removes the first element from A, and returns this element as result. In most imperative settings, as detaching the first element is slow linear operation for array, we can store the trees in descending order by weight, and remove the last element. This is a fast constant time operation. The below C++ example code shows this idea.

```
Node* extract(queue<Node*>& q, Nodes& ts) {
    Node* t;
    if (!q.empty() && (ts.empty() || lessp(q.front(), ts.back()))) {
        t = q.front();
        q.pop();
    } else {
        t = ts.back();
        ts.pop_back();
    }
    return t;
}

Node* huffman2(Nodes ts) {
    queue<Node*> q;
    sort(ts.begin(), ts.end(), greaterp);
    Node* t = extract(q, ts);
    while (!q.empty() || !ts.empty()) {
        q.push(merge(t, extract(q, ts)));
        t = extract(q, ts);
    }
    return t;
```

```
}
```

Note that the sorting isn't necessary if the trees have already been ordered. It can be a linear time reversing in case the trees are in ascending order by weight.

There are three different Huffman man tree building methods explained. Although they follow the same approach developed by Huffman, the result trees varies. Figure 14.48 shows the three different Huffman trees built with these methods.



(a) Created by scan method.

(b) Created by heap method.

(c) Linear time building for sorted list.

Figure 14.48: Variation of Huffman trees for the same symbol list.

Although these three trees are not identical. They are all able to generate the most efficient code. The formal proof is skipped here. The detailed information can be referred to [15] and Section 16.3 of [2].

The Huffman tree building is the core idea of Huffman coding. Many things can be easily achieved with the Huffman tree. For example, the code table can be generated by traversing the tree. We start from the root with the empty prefix p. For any branches, we append a zero to the prefix if turn left, and append a one if turn right. When a leaf node is arrived, the symbol represented by this node and the prefix are put to the code table. Denote the symbol of a leaf node as c, the children for tree T as T_l and T_r respectively. The code table association list can be built with $code(T, \Phi)$, which is defined as below.

$$code(T, p) = \begin{cases} \{(c, p)\} & : & leaf(T) \\ code(T_l, p \cup \{0\}) \cup code(T_r, p \cup \{1\}) & : & otherwise \end{cases} \quad (14.89)$$

Where function $leaf(T)$ tests if tree T is a leaf or a branch node. The

following Haskell example program generates a map as the code table according to this algorithm.

```
code tr = Map.fromList $ traverse [] tr where
  traverse bits (Leaf _ c) = [(c, bits)]
  traverse bits (Branch _ l r) = (traverse (bits ++ [0]) l) ++
                                 (traverse (bits ++ [1]) r)
```

The imperative code table generating algorithm is left as exercise. The encoding process can scan the text, and look up the code table to output the bit sequence. The realization is skipped here.

The decoding process is realized by looking up the Huffman tree according to the bit sequence. We start from the root, whenever a zero is received, we turn left, otherwise if a one is received, we turn right. If a leaf node is arrived, the symbol represented by this leaf is output, and we start another looking up from the root. The decoding process ends when all the bits are consumed. Denote the bit sequence as $B = \{b_1, b_2, ...\}$, all bits except the first one are hold in B', below definition realizes the decoding algorithm.

$$decode(T, B) = \begin{cases} \{c\} & : & B = \Phi \wedge leaf(T) \\ \{c\} \cup decode(root(T), B) & : & leaf(T) \\ decode(T_l, B') & : & b_1 = 0 \\ decode(T_r, B') & : & otherwise \end{cases} \tag{14.90}$$

Where $root(T)$ returns the root of the Huffman tree. The following Haskell example code implements this algorithm.

```
decode tr cs = find tr cs where
  find (Leaf _ c) [] = [c]
  find (Leaf _ c) bs = c : find tr bs
  find (Branch _ l r) (b:bs) = find (if b == 0 then l else r) bs
```

Note that this is an on-line decoding algorithm with linear time performance. It consumes one bit per time. This can be clearly noted from the below imperative realization, where the index keeps increasing by one.

```
1: function DECODE(T, B)
2:     W ← Φ
3:     n ← |B|, i ← 1
4:     while i < n do
5:         R ← T
6:         while ¬ LEAF(R) do
7:             if B[i] = 0 then
8:                 R ← LEFT(R)
9:             else
10:                R ← RIGHT(R)
11:            i ← i + 1
12:        W ← W ∪ SYMBOL(R)
13:    return W
```

This imperative algorithm can be implemented as the following example C++ program.

```
string decode(Node* root, const char* bits) {
```

```
    string w;
    while (*bits) {
        Node* t = root;
        while (!isleaf(t))
            t = '0' == *bits++ ? t->left : t->right;
        w += t->c;
    }
    return w;
}
```

Huffman coding, especially the Huffman tree building shows an interesting strategy. Each time, there are multiple options for merging. Among the trees in the list, Huffman method always selects two trees with the smallest weight. This is the best choice at that merge stage. However, these series of *local* best options generate a global optimal prefix code.

It's not always the case that the local optimal choice also leads to the global optimal solution. In most cases, it doesn't. Huffman coding is a special one. We call the strategy that always choosing the local best option as *greedy* strategy.

Greedy method works for many problems. However, it's not easy to tell if the greedy method can be applied to get the global optimal solution. The generic formal proof is still an active research area. Section 16.4 in [2] provides a good treatment for Matroid tool, which covers many problems that greedy algorithm can be applied.

Change-making problem

We often change money when visiting other countries. People tend to use credit card more often nowadays than before, because it's quite convenient to buy things without considering much about changes. If we changed some money in the bank, there are often some foreign money left by the end of the trip. Some people like to change them to coins for collection. Can we find a solution, which can change the given amount of money with the least number of coins?

Let's use USA coin system for example. There are 5 different coins: 1 cent, 5 cent, 25 cent, 50 cent, and 1 dollar. A dollar is equal to 100 cents. Using the greedy method introduced above, we can always pick the largest coin which is not greater than the remaining amount of money to be changed. Denote list $C = \{1, 5, 25, 50, 100\}$, which stands for the value of coins. For any given money X, the change coins can be generated as below.

$$change(X, C) = \begin{cases} \Phi & : \quad X = 0 \\ \{c_m\} \cup change(X - c_m, C) & : \quad otherwise, \\ & \quad c_m = max(\{c \in C, c \leq X\}) \end{cases} \tag{14.91}$$

If C is in descending order, c_m can be found as the first one not greater than X. If we want to change 1.42 dollar, This function produces a coin list of $\{100, 25, 5, 5, 5, 1, 1\}$. The output coins list can be easily transformed to contain pairs $\{(100, 1), (25, 1), (5, 3), (1, 2)\}$. That we need one dollar, a quarter, three coins of 5 cent, and 2 coins of 1 cent to make the change. The following Haskell example program outputs result as such.

```
solve x = assoc ∘ change x where
```

```
  change 0 _ = []
  change x cs = let c = head $ filter (≤ x) cs in c : change (x - c) cs

assoc = (map (λcs → (head cs, length cs))) ∘ group
```

As mentioned above, this program assumes the coins are in descending order, for instance like below.

```
solve 142 [100, 50, 25, 5, 1]
```

This algorithm is tail recursive, it can be transformed to a imperative looping.

1: **function** CHANGE(X, C)
2: $R \leftarrow \Phi$
3: **while** $X \neq 0$ **do**
4: $c_m = max(\{c \in C, c \leq X\})$
5: $R \leftarrow \{c_m\} \cup R$
6: $X \leftarrow X - c_m$
7: **return** R

The following example Python program implements this imperative version and manages the result with a dictionary.

```
def change(x, coins):
    cs = {}
    while x != 0:
        m = max([c for c in coins if c ≤ x])
        cs[m] = 1 + cs.setdefault(m, 0)
        x = x - m
    return cs
```

For a coin system like USA, the greedy approach can find the optimal solution. The amount of coins is the minimum. Fortunately, our greedy method works in most countries. But it is not always true. For example, suppose a country have coins of value 1, 3, and 4 units. The best change for value 6, is to use two coins of 3 units, however, the greedy method gives a result of three coins: one coin of 4, two coins of 1. Which isn't the optimal result.

Summary of greedy method

As shown in the change making problem, greedy method doesn't always give the best result. In order to find the optimal solution, we need dynamic programming which will be introduced in the next section.

However, the result is often good enough in practice. Let's take the word-wrap problem for example. In modern software editors and browsers, text spans to multiple lines if the length of the content is too long to be hold. With word-wrap supported, user needn't hard line breaking. Although dynamic programming can wrap with the minimum number of lines, it's overkill. On the contrary, greedy algorithm can wrap with lines approximate to the optimal result with quite effective realization as below. Here it wraps text T, not to exceeds line width W, with space s between each word.

1: $L \leftarrow W$
2: **for** $w \in T$ **do**
3: **if** $|w| + s > L$ **then**

```
4:          Insert line break
5:          L ← W − |w|
6:      else
7:          L ← L − |w| − s
```

For each word w in the text, it uses a greedy strategy to put as many words in a line as possible unless it exceeds the line width. Many word processors use a similar algorithm to do word-wrapping.

There are many cases, the strict optimal result, but not the approximate one is necessary. Dynamic programming can help to solve such problems.

Dynamic programming

In the change-making problem, we mentioned the greedy method can't always give the optimal solution. For any coin system, are there any way to find the best changes?

Suppose we have find the best solution which makes X value of money. The coins needed are contained in C_m. We can partition these coins into two collections, C_1 and C_2. They make money of X_1, and X_2 respectively. We'll prove that C_1 is the optimal solution for X_1, and C_2 is the optimal solution for X_2.

Proof. For X_1, Suppose there exists another solution C_1', which uses less coins than C_1. Then changing solution $C_1' \cup C_2$ uses less coins to make X than C_m. This is conflict with the fact that C_m is the optimal solution to X. Similarity, we can prove C_2 is the optimal solution to X_2. □

Note that it is not true in the reverse situation. If we arbitrary select a value $Y < X$, divide the original problem to find the optimal solutions for sub problems Y and $X - Y$. Combine the two optimal solutions doesn't necessarily yield optimal solution for X. Consider this example. There are coins with value 1, 2, and 4. The optimal solution for making value 6, is to use 2 coins of value 2, and 4; However, if we divide $6 = 3 + 3$, since each 3 can be made with optimal solution $3 = 1 + 2$, the combined solution contains 4 coins $(1 + 1 + 2 + 2)$.

If an optimal problem can be divided into several sub optimal problems, we call it has optimal substructure. We see that the change-making problem has optimal substructure. But the dividing has to be done based on the coins, but not with an arbitrary value.

The optimal substructure can be expressed recursively as the following.

$$change(X) = \begin{cases} \Phi & : \quad X = 0 \\ least(\{c \cup change(X - c) | c \in C, c \leq X\}) & : \quad otherwise \end{cases} \tag{14.92}$$

For any coin system C, the changing result for zero is empty; otherwise, we check every candidate coin c, which is not greater then value X, and recursively find the best solution for $X - c$; We pick the coin collection which contains the least coins as the result.

Below Haskell example program implements this top-down recursive solution.

```
change _ 0 = []
change cs x = minimumBy (compare `on` length)
                [c:change cs (x - c) | c ← cs, c ≤ x]
```

Although this program outputs correct answer `[2, 4]` when evaluates `change [1, 2, 4] 6`, it performs very bad when changing 1.42 dollar with USA coins system. It failed to find the answer within 15 minutes in a computer with 2.7GHz CPU and 8G memory.

The reason why it's slow is because there are a lot of duplicated computing in the top-down recursive solution. When it computes $change(142)$, it needs to examine $change(141), change(137), change(117), change(92)$, and $change(42)$. While $change(141)$ next computes to smaller values by deducing with 1, 2, 25, 50 and 100 cents. it will eventually meets value 137, 117, 92, and 42 again. The search space explodes with power of 5.

This is quite similar to compute Fibonacci numbers in a top-down recursive way.

$$F_n = \begin{cases} 1 & : \quad n = 1 \vee n = 2 \\ F_{n-1} + F_{n-2} & : \quad otherwise \end{cases} \tag{14.93}$$

When we calculate F_8 for example, we recursively calculate F_7 and F_6. While when we calculate F_7, we need calculate F_6 again, and F_5, ... As shown in the below expand forms, the calculation is doubled every time, and the same value is calculate again and again.

$$\begin{aligned} F_8 &= F_7 + F_6 \\ &= F_6 + F_5 + F_5 + F_4 \\ &= F_5 + F_4 + F_4 + F_3 + F_4 + F_3 + F_3 + F_2 \\ &= ... \end{aligned}$$

In order to avoid duplicated computation, a table F can be maintained when calculating the Fibonacci numbers. The first two elements are filled as 1, all others are left blank. During the top-down recursive calculation, If need F_k, we first look up this table for the k-th cell, if it isn't blank, we use that value directly. Otherwise we need further calculation. Whenever a value is calculated, we store it in the corresponding cell for looking up in the future.

```
1: F ← {1, 1, NIL, NIL, ...}
2: function FIBONACCI(n)
3:     if n > 2 ∧ F[n] = NIL then
4:         F[n] ← FIBONACCI(n − 1) + FIBONACCI(n − 2)
5:     return F[n]
```

By using the similar idea, we can develop a new top-down change-making solution. We use a table T to maintain the best changes, it is initialized to all empty coin list. During the top-down recursive computation, we look up this table for smaller changing values. Whenever a intermediate value is calculated, it is stored in the table.

```
1: T ← {Φ, Φ, ...}
2: function CHANGE(X)
3:     if X > 0 ∧ T[X] = Φ then
4:         for c ∈ C do
5:             if c ≤ X then
```

```
6:              C_m ← {c} ∪ CHANGE(X − c)
7:              if T[X] = Φ ∨ |C_m| < |T[X]| then
8:                  T[X] ← C_m
9:      return T[X]
```

The solution to change 0 money is definitely empty Φ, otherwise, we look up $T[X]$ to retrieve the solution to change X money. If it is empty, we need recursively calculate it. We examine all coins in the coin system C which is not greater than X. This is the sub problem of making changes for money $X - c$. The minimum amount of coins plus one coin of c is stored in $T[X]$ finally as the result.

The following example Python program implements this algorithm just takes 8000 ms to give the answer of changing 1.42 dollar in US coin system.

```
tab = [[] for _ in range(1000)]

def change(x, cs):
    if x > 0 and tab[x] == []:
        for s in [[c] + change(x - c, cs) for c in cs if c <= x]:
            if tab[x] == [] or len(s) < len(tab[x]):
                tab[x] = s
    return tab[x]
```

Another solution to calculate Fibonacci number, is to compute them in order of $F_1, F_2, F_3, ..., F_n$. This is quite natural when people write down Fibonacci series.

```
1: function FIBO(n)
2:     F = {1, 1, NIL, NIL, ...}
3:     for i ← 3 to n do
4:         F[i] ← F[i − 1] + F[i − 2]
5:     return F[n]
```

We can use the quite similar idea to solve the change making problem. Starts from zero money, which can be changed by an empty list of coins, we next try to figure out how to change money of value 1. In US coin system for example, A cent can be used; The next values of 2, 3, and 4, can be changed by two coins of 1 cent, three coins of 1 cent, and 4 coins of 1 cent. At this stage, the solution table looks like below

0	1	2	3	4
Φ	$\{1\}$	$\{1,1\}$	$\{1,1,1\}$	$\{1,1,1,1\}$

The interesting case happens for changing value 5. There are two options, use another coin of 1 cent, which need 5 coins in total; The other way is to use 1 coin of 5 cent, which uses less coins than the former. So the solution table can be extended to this.

0	1	2	3	4	5
Φ	$\{1\}$	$\{1,1\}$	$\{1,1,1\}$	$\{1,1,1,1\}$	$\{5\}$

For the next change value 6, since there are two types of coin, 1 cent and 5 cent, are less than this value, we need examine both of them.

- If we choose the 1 cent coin, we need next make changes for 5; Since we've already known that the best solution to change 5 is $\{5\}$, which only needs a coin of 5 cents, by looking up the solution table, we have one candidate solution to change 6 as $\{5, 1\}$;

- The other option is to choose the 5 cent coin, we need next make changes for 1; By looking up the solution table we've filled so far, the sub optimal solution to change 1 is $\{1\}$. Thus we get another candidate solution to change 6 as $\{1, 5\}$;

It happens that, both options yield a solution of two coins, we can select either of them as the best solution. Generally speaking, the candidate with fewest number of coins is selected as the solution, and filled into the table.

At any iteration, when we are trying to change the $i < X$ value of money, we examine all the types of coin. For any coin c not greater than i, we look up the solution table to fetch the sub solution $T[i - c]$. The number of coins in this sub solution plus the one coin of c are the total coins needed in this candidate solution. The fewest candidate is then selected and updated to the solution table.

The following algorithm realizes this bottom-up idea.

```
1: function CHANGE(X)
2:     T ← {Φ, Φ, ...}
3:     for i ← 1 to X do
4:         for c ∈ C, c ≤ i do
5:             if T[i] = Φ ∨ 1 + |T[i − c]| < |T[i]| then
6:                 T[i] ← {c} ∪ T[i − c]
7:     return T[X]
```

This algorithm can be directly translated to imperative programs, like Python for example.

```
def changemk(x, cs):
    s = [[] for _ in range(x+1)]
    for i in range(1, x+1):
        for c in cs:
            if c ≤ i and (s[i] == [] or 1 + len(s[i-c]) < len(s[i])):
                s[i] = [c] + s[i-c]
    return s[x]
```

Observe the solution table, it's easy to find that, there are many duplicated contents being stored.

6	7	8	9	10	...
$\{1, 5\}$	$\{1, 1, 5\}$	$\{1, 1, 1, 5\}$	$\{1, 1, 1, 1, 5\}$	$\{5, 5\}$	...

This is because the optimal sub solutions are completely copied and saved in parent solution. In order to use less space, we can only record the 'delta' part from the sub optimal solution. In change-making problem, it means that we only need to record the coin being selected for value i.

```
1: function CHANGE'(X)
2:     T ← {0, ∞, ∞, ...}
3:     S ← {NIL, NIL, ...}
4:     for i ← 1 to X do
5:         for c ∈ C, c ≤ i do
6:             if 1 + T[i − c] < T[i] then
7:                 T[i] ← 1 + T[i − c]
8:                 S[i] ← c
9:     while X > 0 do
10:        PRINT(S[X])
```

11: $X \leftarrow X - S[X]$

Instead of recording the complete solution list of coins, this new algorithm uses two tables T and S. T holds the minimum number of coins needed for changing value 0, 1, 2, ...; while S holds the first coin being selected for the optimal solution. For the complete coin list to change money X, the first coin is thus $S[X]$, the sub optimal solution is to change money $X' = X - S[X]$. We can look up table $S[X']$ for the next coin. The coins for sub optimal solutions are repeatedly looked up like this till the beginning of the table. Below Python example program implements this algorithm.

```
def chgmk(x, cs):
    cnt = [0] + [x+1] * x
    s = [0]
    for i in range(1, x+1):
        coin = 0
        for c in cs:
            if c ≤ i and 1 + cnt[i-c] < cnt[i]:
                cnt[i] = 1 + cnt[i-c]
                coin = c
        s.append(coin)
    r = []
    while x > 0:
        r.append(s[x])
        x = x - s[x]
    return r
```

This change-making solution loops n times for given money n. It examines at most the full coin system in each iteration. The time is bound to $\Theta(nk)$ where k is the number of coins for a certain coin system. The last algorithm adds $O(n)$ spaces to record sub optimal solutions with table T and S.

In purely functional settings, There is no means to mutate the solution table and look up in constant time. One alternative is to use finger tree as we mentioned in previous chapter [11]. We can store the minimum number of coins, and the coin leads to the sub optimal solution in pairs.

The solution table, which is a finger tree, is initialized as $T = \{(0, 0)\}$. It means change 0 money need no coin. We can fold on list $\{1, 2, ..., X\}$, start from this table, with a binary function $change(T, i)$. The folding will build the solution table, and we can construct the coin list from this table by function $make(X, T)$.

$$makeChange(X) = make(X, fold(change, \{(0, 0)\}, \{1, 2, ..., X\})) \quad (14.94)$$

In function $change(T, i)$, all the coins not greater than i are examined to select the one lead to the best result. The fewest number of coins, and the coin being selected are formed to a pair. This pair is inserted to the finger tree, so that a new solution table is returned.

$$change(T, i) = insert(T, fold(sel, (\infty, 0), \{c | c \in C, c \leq i\})) \quad (14.95)$$

[11] Some purely functional programming environments, Haskell for instance, provide built-in array; while other almost pure ones, such as ML, provide mutable array

Again, folding is used to select the candidate with the minimum number of coins. This folding starts with initial value $(\infty, 0)$, on all valid coins. function $sel((n, c), c')$ accepts two arguments, one is a pair of length and coin, which is the best solution so far; the other is a candidate coin, it examines if this candidate can make better solution.

$$sel((n, c), c') = \begin{cases} (1 + n', c') & : \quad 1 + n' < n, (n', c') = T[i - c'] \\ (n, c) & : \quad otherwise \end{cases} \tag{14.96}$$

After the solution table is built, the coins needed can be generated from it.

$$make(X, T) = \begin{cases} \Phi & : \quad X = 0 \\ \{c\} \cup make(X - c, T) & : \quad otherwise, (n, c) = T[X] \end{cases} \tag{14.97}$$

The following example Haskell program uses `Data.Sequence`, which is the library of finger tree, to implement change making solution.

```
import Data.Sequence (Seq, singleton, index, (|>))

changemk x cs = makeChange x $ foldl change (singleton (0, 0)) [1..x] where
  change tab i = let sel c = min (1 + fst (index tab (i - c)), c)
                 in tab |> (foldr sel ((x + 1), 0) $ filter (≤i) cs)
  makeChange 0 _ = []
  makeChange x tab = let c = snd $ index tab x in c : makeChange (x - c) tab
```

It's necessary to memorize the optimal solution to sub problems no matter using the top-down or the bottom-up approach. This is because a sub problem is used many times when computing the overall optimal solution. Such properties are called overlapping sub problems.

Properties of dynamic programming

Dynamic programming was originally named by Richard Bellman in 1940s. It is a powerful tool to search for optimal solution for problems with two properties.

- Optimal sub structure. The problem can be broken down into smaller problems, and the optimal solution can be constructed efficiently from solutions of these sub problems;
- Overlapping sub problems. The problem can be broken down into sub problems which are reused several times in finding the overall solution.

The change-making problem, as we've explained, has both optimal sub structures, and overlapping sub problems.

Longest common subsequence problem

The longest common subsequence problem, is different with the longest common substring problem. We've show how to solve the later in the chapter of suffix tree. The longest common subsequence needn't be consecutive part of the original sequence.

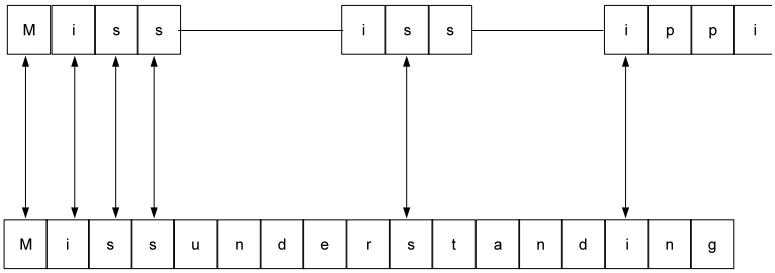


Figure 14.49: The longest common subsequence

For example, The longest common substring for text "Mississippi", and "Missunderstanding" is "Miss", while the longest common subsequence for them are "Misssi". This is shown in figure 14.49.

If we rotate the figure vertically, and consider the two texts as two pieces of source code, it turns to be a 'diff' result between them. Most modern version control tools need calculate the difference content among the versions. The longest common subsequence problem plays a very important role.

If either one of the two strings X and Y is empty, the longest common subsequence $LCS(X, Y)$ is definitely empty; Otherwise, denote $X = \{x_1, x_2, ..., x_n\}$, $Y = \{y_1, y_2, ..., y_m\}$, if the first elements x_1 and y_1 are same, we can recursively find the longest subsequence of $X' = \{x_2, x_3, ..., x_n\}$ and $Y' = \{y_2, y_3, ..., y_m\}$. And the final result $LCS(X, Y)$ can be constructed by concatenating x_1 with $LCS(X', Y')$; Otherwise if $x_1 \neq y_1$, we need recursively find the longest common subsequences of $LCS(X, Y')$ and $LCS(X', Y)$, and pick the longer one as the final result. Summarize these cases gives the below definition.

$$LCS(X, Y) = \begin{cases} \Phi & : \quad X = \Phi \vee Y = \Phi \\ \{x_1\} \cup LCS(X', Y') & : \quad x_1 = y_1 \\ longer(LCS(X, Y'), LCS(X', Y)) & : \quad otherwise \end{cases} \tag{14.98}$$

Note that this algorithm shows clearly the optimal substructure, that the longest common subsequence problem can be broken to smaller problems. The sub problem is ensured to be at least one element shorter than the original one.

It's also clear that, there are overlapping sub-problems. The longest common subsequences to the sub strings are used multiple times in finding the overall optimal solution.

The existence of these two properties, the optimal substructure and the overlapping sub-problem, indicates the dynamic programming can be used to

solve this problem.

A 2-dimension table can be used to record the solutions to the sub-problems. The rows and columns represent the substrings of X and Y respectively.

		a	n	t	e	n	n	a
		1	2	3	4	5	6	7
b	1							
a	2							
n	3							
a	4							
n	5							
a	6							

This table shows an example of finding the longest common subsequence for strings "antenna" and "banana". Their lengths are 7, and 6. The right bottom corner of this table is looked up first, Since it's empty we need compare the 7th element in "antenna" and the 6th in "banana", they are both 'a', Thus we need next recursively look up the cell at row 5, column 6; It's still empty, and we repeated this till either get a trivial case that one substring becomes empty, or some cell we are looking up has been filled before. Similar to the change-making problem, whenever the optimal solution for a sub-problem is found, it is recorded in the cell for further reusing. Note that this process is in the reversed order comparing to the recursive equation given above, that we start from the right most element of each string.

Considering that the longest common subsequence for any empty string is still empty, we can extended the solution table so that the first row and column hold the empty strings.

		a	n	t	e	n	n	a
		Φ	Φ	Φ	Φ	Φ	Φ	Φ
b	Φ							
a	Φ							
n	Φ							
a	Φ							
n	Φ							
a	Φ							

Below algorithm realizes the top-down recursive dynamic programming solution with such a table.

```
1: T ← NIL
2: function LCS(X, Y)
3:     m ← |X|, n ← |Y|
4:     m' ← m + 1, n' ← n + 1
5:     if T = NIL then
6:         T ← {{Φ, Φ, ..., Φ}, {Φ, NIL, NIL, ...}, ...}                ▷ m' × n'
7:     if X ≠ Φ ∧ Y ≠ Φ ∧ T[m'][n'] = NIL then
8:         if X[m] = Y[n] then
9:             T[m'][n'] ← APPEND(LCS(X[1..m − 1], Y[1..n − 1]), X[m])
10:        else
11:            T[m'][n'] ← LONGER(LCS(X, Y[1..n − 1]), LCS(X[1..m − 1], Y))
12:    return T[m'][n']
```

The table is firstly initialized with the first row and column filled with empty

strings; the rest are all NIL values. Unless either string is empty, or the cell content isn't NIL, the last two elements of the strings are compared, and recursively computes the longest common subsequence with substrings. The following Python example program implements this algorithm.

```
def lcs(xs, ys):
    m = len(xs)
    n = len(ys)
    global tab
    if tab is None:
        tab = [[""]*(n+1)] + [[""] + [None]*n for _ in xrange(m)]
    if m != 0 and n !=0 and tab[m][n] is None:
        if xs[-1] == ys[-1]:
            tab[m][n] = lcs(xs[:-1], ys[:-1]) + xs[-1]
        else:
            (a, b) = (lcs(xs, ys[:-1]), lcs(xs[:-1], ys))
            tab[m][n] = a if len(b) < len(a) else b
    return tab[m][n]
```

The longest common subsequence can also be found in a bottom-up manner as what we've done with the change-making problem. Besides that, instead of recording the whole sequences in the table, we can just store the lengths of the longest subsequences, and later construct the subsubsequence with this table and the two strings. This time, the table is initialized with all values set as 0.

```
1: function LCS(X, Y)
2:     m ← |X|, n ← |Y|
3:     T ← {{0, 0, ...}, {0, 0, ...}, ...}                ▷ (m + 1) × (n + 1)
4:     for i ← 1 to m do
5:         for j ← 1 to n do
6:             if X[i] = Y[j] then
7:                 T[i + 1][j + 1] ← T[i][j] + 1
8:             else
9:                 T[i + 1][j + 1] ← MAX(T[i][j + 1], T[i + 1][j])
10:    return GET(T, X, Y, m, n)

11: function GET(T, X, Y, i, j)
12:    if i = 0 ∨ j = 0 then
13:        return Φ
14:    else if X[i] = Y[j] then
15:        return APPEND(GET(T, X, Y, i − 1, j − 1), X[i])
16:    else if T[i − 1][j] > T[i][j − 1] then
17:        return GET(T, X, Y, i − 1, j)
18:    else
19:        return GET(T, X, Y, i, j − 1)
```

In the bottom-up approach, we start from the cell at the second row and the second column. The cell is corresponding to the first element in both X, and Y. If they are same, the length of the longest common subsequence so far is 1. This can be yielded by increasing the length of empty sequence, which is stored in the top-left cell, by one; Otherwise, we pick the maximum value from the upper cell and left cell. The table is repeatedly filled in this manner.

After that, a back-track is performed to construct the longest common sub-

sequence. This time we start from the bottom-right corner of the table. If the last elements in X and Y are same, we put this element as the last one of the result, and go on looking up the cell along the diagonal line; Otherwise, we compare the values in the left cell and the right cell, and go on looking up the cell with the bigger value.

The following example Python program implements this algorithm.

```
def lcs(xs, ys):
    m = len(xs)
    n = len(ys)
    c = [[0]*(n+1) for _ in xrange(m+1)]
    for i in xrange(1, m+1):
        for j in xrange(1, n+1):
            if xs[i-1] == ys[j-1]:
                c[i][j] = c[i-1][j-1] + 1
            else:
                c[i][j] = max(c[i-1][j], c[i][j-1])

    return get(c, xs, ys, m, n)

def get(c, xs, ys, i, j):
    if i==0 or j==0:
        return []
    elif xs[i-1] == ys[j-1]:
        return get(c, xs, ys, i-1, j-1) + [xs[i-1]]
    elif c[i-1][j] > c[i][j-1]:
        return get(c, xs, ys, i-1, j)
    else:
        return get(c, xs, ys, i, j-1)
```

The bottom-up dynamic programming solution can also be defined in purely functional way. The finger tree can be used as a table. The first row is filled with $n+1$ zero values. This table can be built by folding on sequence X. Then the longest common subsequence is constructed from the table.

$$LCS(X, Y) = construct(fold(f, \{\{0, 0, ..., 0\}\}, zip(\{1, 2, ...\}, X))) \tag{14.99}$$

Note that, since the table need be looked up by index, X is zipped with natural numbers. Function f creates a new row of this table by folding on sequence Y, and records the lengths of the longest common sequence for all possible cases so far.

$$f(T, (i, x)) = insert(T, fold(longest, \{0\}, zip(\{1, 2, ...\}, Y))) \tag{14.100}$$

Function *longest* takes the intermediate filled row result, and a pair of index and element in Y, it compares if this element is the same as the one in X. Then fills the new cell with the length of the longest one.

$$longest(R, (j, y)) = \begin{cases} insert(R, 1 + T[i-1][j-1]) & : \quad x = y \\ insert(R, max(T[i-1][j], T[i][j-1])) & : \quad otherwise \end{cases} \tag{14.101}$$

After the table is built. The longest common sub sequence can be constructed recursively by looking up this table. We can pass the reversed sequences $\overleftarrow{X}$, and $\overleftarrow{Y}$ together with their lengths m and n for efficient building.

$$construct(T) = get((\overleftarrow{X}, m), (\overleftarrow{Y}, n)) \tag{14.102}$$

If the sequences are not empty, denote the first elements as x and y. The rest elements are hold in $\overleftarrow{X'}$ and $\overleftarrow{Y'}$ respectively. The function get can be defined as the following.

$$get((\overleftarrow{X}, i), (\overleftarrow{Y}, j)) = \begin{cases} \Phi & : \overleftarrow{X} = \Phi \wedge \overleftarrow{Y} = \Phi \\ get((\overleftarrow{X'}, i-1), (\overleftarrow{Y'}, j-1)) \cup \{x\} & : x = y \\ get((\overleftarrow{X'}, i-1), (\overleftarrow{Y}, j)) & : T[i-1][j] > T[i][j-1] \\ get((\overleftarrow{X}, i), (\overleftarrow{Y'}, j-1)) & : otherwise \end{cases} \tag{14.103}$$

Below Haskell example program implements this solution.

```
lcs' xs ys = construct $ foldl f (singleton $ fromList $ replicate (n+1) 0)
                                 (zip [1..] xs) where
  (m, n) = (length xs, length ys)
  f tab (i, x) = tab |> (foldl longer (singleton 0) (zip [1..] ys)) where
    longer r (j, y) = r |> if x == y
                      then 1 + (tab `index` (i-1) `index` (j-1))
                      else max (tab `index` (i-1) `index` j) (r `index` (j-1))
  construct tab = get (reverse xs, m) (reverse ys, n) where
    get ([], 0) ([], 0) = []
    get ((x:xs), i) ((y:ys), j)
      | x == y = get (xs, i-1) (ys, j-1) ++ [x]
      | (tab `index` (i-1) `index` j) > (tab `index` i `index` (j-1)) =
                get (xs, i-1) ((y:ys), j)
      | otherwise = get ((x:xs), i) (ys, j-1)
```

Subset sum problem

Dynamic programming does not limit to solve the optimization problem, but can also solve some more general searching problems. Subset sum problem is such an example. Given a set of integers, is there a non-empty subset sums to zero? for example, there are two subsets of $\{11, 64, -82, -68, 86, 55, -88, -21, 51\}$ both sum to zero. One is $\{64, -82, 55, -88, 51\}$, the other is $\{64, -82, -68, 86\}$.

Of course summing to zero is a special case, because sometimes, people want to find a subset, whose sum is a given value s. Here we are going to develop a method to find all the candidate subsets.

There is obvious a brute-force exhausting search solution. For every element, we can either pick it or not. So there are total 2^n options for set with n elements. Because for every selection, we need check if it sums to s. This is a linear operation. The overall complexity is bound to $O(n2^n)$. This is the exponential algorithm, which takes very huge time if the set is big.

There is a recursive solution to subset sum problem. If the set is empty, there is no solution definitely; Otherwise, let the set is $X = \{x_1, x_2, ...\}$. If $x_1 = s$, then subset $\{x_1\}$ is a solution, we need next search for subsets $X' = \{x_2, x_3, ...\}$ for

those sum to s; Otherwise if $x_1 \neq s$, there are two different kinds of possibilities. We need search X' for both sum s, and sum $s - x_1$. For any subset sum to $s - x_1$, we can add x_1 to it to form a new set as a solution. The following equation defines this algorithm.

$$solve(X, s) = \begin{cases} \Phi & : & X = \Phi \\ \{\{x_1\}\} \cup solve(X', s) & : & x_1 = s \\ solve(X', s) \cup \{\{x_1\} \cup S | S \in solve(X', s - x_1)\} & : & otherwise \end{cases} \tag{14.104}$$

There are clear substructures in this definition, although they are not in a sense of optimal. And there are also overlapping sub-problems. This indicates the problem can be solved with dynamic programming with a table to memorize the solutions to sub-problems.

Instead of developing a solution to output all the subsets directly, let's consider how to give the existence answer firstly. That output 'yes' if there exists some subset sum to s, and 'no' otherwise.

One fact is that, the upper and lower limit for all possible answer can be calculated in one scan. If the given sum s doesn't belong to this range, there is no solution obviously.

$$\begin{cases} s_l = \sum\{x \in X, x < 0\} \\ s_u = \sum\{x \in X, x > 0\} \end{cases} \tag{14.105}$$

Otherwise, if $s_l \leq s \leq s_u$, since the values are all integers, we can use a table, with $s_u - s_l + 1$ columns, each column represents a possible value in this range, from s_l to s_u. The value of the cell is either true or false to represents if there exists subset sum to this value. All cells are initialized as false. Starts from the first element x_1 in X, definitely, set $\{x_1\}$ can sum to x_1, so that the cell represents this value in the first row can be filled as true.

	s_l	$s_l + 1$	...	x_1	...	s_u
x_1	F	F	...	T	...	F

With the next element x_2, There are three possible sums. Similar as the first row, $\{x_2\}$ sums to x_2; For all possible sums in previous row, they can also been achieved without x_2. So the cell below to x_1 should also be filled as true; By adding x_2 to all possible sums so far, we can also get some new values. That the cell represents $x_1 + x_2$ should be true.

	s_l	$s_l + 1$	...	x_1	...	x_2	...	$x_1 + x_2$	...	s_u
x_1	F	F	...	T	...	F	...	F	...	F
x_2	F	F	...	T	...	T	...	T	...	F

Generally speaking, when fill the i-th row, all the possible sums constructed with $\{x_1, x_2, ..., x_{i-1}$ so far can also be achieved with x_i. So the cells previously are true should also be true in this new row. The cell represents value x_i should also be true since the singleton set $\{x_i\}$ sums to it. And we can also adds x_i to all previously constructed sums to get the new results. Cells represent these new sums should also be filled as true.

When all the elements are processed like this, a table with $|X|$ rows is built. Looking up the cell represents s in the last row tells if there exists subset can sum to this value. As mentioned above, there is no solution if $s < s_l$ or $s_u < s$. We skip handling this case for the sake of brevity.

1: **function** SUBSET-SUM(X, s)

```
2:    s_l ← Σ{x ∈ X, x < 0}
3:    s_u ← Σ{x ∈ X, x > 0}
4:    n ← |X|
5:    T ← {{False, False, ...}, {False, False, ...}, ...}        ▷ n × (s_u − s_l + 1)
6:    for i ← 1 to n do
7:        for j ← s_l to s_u do
8:            if X[i] = j then
9:                T[i][j] ← True
10:           if i > 1 then
11:               T[i][j] ← T[i][j] ∨ T[i − 1][j]
12:               j' ← j − X[i]
13:               if s_l ≤ j' ≤ s_u then
14:                   T[i][j] ← T[i][j] ∨ T[i − 1][j']
15:   return T[n][s]
```

Note that the index to the columns of the table, doesn't range from 1 to $s_u - s_l + 1$, but maps directly from s_l to s_u. Because most programming environments don't support negative index, this can be dealt with $T[i][j - s_l]$. The following example Python program utilizes the property of negative indexing.

```
def solve(xs, s):
    low = sum([x for x in xs if x < 0])
    up  = sum([x for x in xs if x > 0])
    tab = [[False]*(up-low+1) for _ in xs]
    for i in xrange(0, len(xs)):
        for j in xrange(low, up+1):
            tab[i][j] = (xs[i] == j)
            j1 = j - xs[i];
            tab[i][j] = tab[i][j] or tab[i-1][j] or
                        (low <= j1 and j1 <= up and tab[i-1][j1])
    return tab[-1][s]
```

Note that this program doesn't use different branches for $i = 0$ and $i = 1, 2, ..., n - 1$. This is because when $i = 0$, the row index to $i - 1 = -1$ refers to the last row in the table, which are all false. This simplifies the logic one more step.

With this table built, it's easy to construct all subsets sum to s. The method is to look up the last row for cell represents s. If the last element $x_n = s$, then $\{x_n\}$ definitely is a candidate. We next look up the previous row for s, and recursively construct all the possible subsets sum to s with $\{x_1, x_2, x_3, ..., x_{n-1}\}$. Finally, we look up the second last row for cell represents $s - x_n$. And for every subset sums to this value, we add element x_n to construct a new subset, which sums to s.

```
1: function GET(X, s, T, n)
2:     S ← Φ
3:     if X[n] = s then
4:         S ← S ∪ {X[n]}
5:     if n > 1 then
6:         if T[n − 1][s] then
7:             S ← S∪ GET(X, s, T, n − 1)
8:         if T[n − 1][s − X[n]] then
```

```
9:          S ← S ∪ {{X[n]} ∪ S' | S' ∈ GET(X, s − X[n], T, n − 1) }
10:     return S
```

The following Python example program translates this algorithm.

```
def get(xs, s, tab, n):
    r = []
    if xs[n] == s:
        r.append([xs[n]])
    if n > 0:
        if tab[n-1][s]:
            r = r + get(xs, s, tab, n-1)
        if tab[n-1][s - xs[n]]:
            r = r + [[xs[n]] + ys for ys in get(xs, s - xs[n], tab, n-1)]
    return r
```

This dynamic programming solution to subset sum problem loops $O(n(s_u - s_l + 1))$ times to build the table, and recursively uses $O(n)$ time to construct the final solution from this table. The space it used is also bound to $O(n(s_u - s_l + 1))$.

Instead of using table with n rows, a vector can be used alternatively. For every cell represents a possible sum, the list of subsets are stored. This vector is initialized to contain all empty sets. For every element in X, we update the vector, so that it records all the possible sums which can be built so far. When all the elements are considered, the cell corresponding to s contains the final result.

```
1:  function SUBSET-SUM(X, s)
2:      s_l ← Σ{x ∈ X, x < 0}
3:      s_u ← Σ{x ∈ X, x > 0}
4:      T ← {Φ, Φ, ...}                                  ▷ s_u − s_l + 1
5:      for x ∈ X do
6:          T' ← DUPLICATE(T)
7:          for j ← s_l to s_u do
8:              j' ← j − x
9:              if x = j then
10:                 T'[j] ← T'[j] ∪ {x}
11:             if s_l ≤ j' ≤ s_u ∧ T[j'] ≠ Φ then
12:                 T'[j] ← T'[j] ∪ {{x} ∪ S | S ∈ T[j']}
13:         T ← T'
14:     return T[s]
```

The corresponding Python example program is given as below.

```
def subsetsum(xs, s):
    low = sum([x for x in xs if x < 0])
    up  = sum([x for x in xs if x > 0])
    tab = [[] for _ in xrange(low, up+1)]
    for x in xs:
        tab1 = tab[:]
        for j in xrange(low, up+1):
            if x == j:
                tab1[j].append([x])
            j1 = j - x
            if low ≤ j1 and j1 ≤ up and tab[j1] != []:
                tab1[j] = tab1[j] + [[x] + ys for ys in tab[j1]]
```

```
        tab = tab1
    return tab[s]
```

This imperative algorithm shows a clear structure, that the solution table is built by looping every element. This can be realized in purely functional way by folding. A finger tree can be used to represents the vector spans from s_l to s_u. It is initialized with all empty values as in the following equation.

$$subsetsum(X, s) = fold(build, \{\Phi, \Phi, ..., \}, X)[s] \tag{14.106}$$

After folding, the solution table is built, the answer is looked up at cell s[12].

For every element $x \in X$, function *build* folds the list $\{s_l, s_l + 1, ..., s_u\}$, with every value j, it checks if it equals to x and appends the singleton set $\{x\}$ to the j-th cell. Not that here the cell is indexed from s_l, but not 0. If the cell corresponding to $j - x$ is not empty, the candidate solutions stored in that place are also duplicated and add element x is added to every solution.

$$build(T, x) = fold(f, T, \{s_l, s_l + 1, ..., s_u\}) \tag{14.107}$$

$$f(T, j) = \begin{cases} T'[j] \cup \{\{x\} \cup Y | Y \in T[j']\} & : \quad s_l \le j' \le s_u \wedge T[j'] \neq \Phi, j' = j - x \\ T' & : \quad otherwise \end{cases} \tag{14.108}$$

Here the adjustment is applied on T', which is another adjustment to T as shown as below.

$$T' = \begin{cases} \{x\} \cup T[j] & : \quad x = j \\ T & : \quad otherwise \end{cases} \tag{14.109}$$

Note that the first clause in both equation (14.108) and (14.109) return a new table with certain cell being updated with the given value.

The following Haskell example program implements this algorithm.

```
subsetsum xs s = foldl build (fromList [[] | _ ← [1..u]]) xs `idx` s where
  l = sum $ filter (< 0) xs
  u = sum $ filter (> 0) xs
  idx t i = index t (i - l)
  build tab x = foldl (λt j → let j' = j - x in
                  adjustIf (l ≤ j' && j' ≤ u && tab `idx` j' /= [])
                           (++ [(x:ys) | ys ← tab `idx` j']) j
                           (adjustIf (x == j) ([x]:) j t)) tab [l..u]
  adjustIf pred f i seq = if pred then adjust f (i - l) seq else seq
```

Some materials like [16] provide common structures to abstract dynamic programming. So that problems can be solved with a generic solution by customizing the precondition, the comparison of candidate solutions for better choice, and the merge method for sub solutions. However, the variety of problems makes things complex in practice. It's important to study the properties of the problem carefully.

Exercise 14.3

[12] Again, here we skip the error handling to the case that $s < s_l$ or $s > s_u$. There is no solution if s is out of range.

- Realize a maze solver by using the stack approach, which can find all the possible paths.
- There are 92 distinct solutions for the 8 queens puzzle. For any one solution, rotating it $90°, 180°, 270°$ gives solutions too. Also flipping it vertically and horizontally also generate solutions. Some solutions are symmetric, so that rotation or flip gives the same one. There are 12 unique solutions in this sense. Modify the program to find the 12 unique solutions. Improve the program, so that the 92 distinct solutions can be found with fewer search.
- Make the 8 queens puzzle solution generic so that it can solve n queens puzzle.
- Make the functional solution to the leap frogs puzzle generic, so that it can solve n frogs case.
- Modify the wolf, goat, and cabbage puzzle algorithm, so that it can find all possible solutions.
- Give the complete algorithm definition to solve the 2 water jugs puzzle with extended Euclid algorithm.
- We needn't the exact linear combination information x and y in fact. After we know the puzzle is solvable by testing with GCD, we can blindly execute the process that: fill A, pour A into B, whenever B is full, empty it till there is expected volume in one jug. Realize this solution. Can this one find faster solution than the original version?
- Compare to the extended Euclid method, the DFS approach is a kind of brute-force searching. Improve the extended Euclid approach by finding the best linear combination which minimize $|x| + |y|$.
- Realize the imperative Huffman code table generating algorithm.
- One option to realize the bottom-up solution for the longest common subsequence problem is to record the direction in the table. Thus, instead of storing the length information, three values like 'N', for north, 'W' for west, and 'NW' for northwest are used to indicate how to construct the final result. We start from the bottom-right corner of the table, if the cell value is 'NW', we go along the diagonal by moving to the cell in the upper-left; if it's 'N', we move vertically to the upper row; and move horizontally if it's 'W'. Implement this approach in your favorite programming language.

14.4 Short summary

This chapter introduces the elementary methods about searching. Some of them instruct the computer to scan for interesting information among the data. They often have some structure, that can be updated during the scan. This can be considered as a special case for the information reusing approach. The other commonly used strategy is divide and conquer, that the scale of the search

domain is kept decreasing till some obvious result. This chapter also explains methods to search for solutions among domains. The solutions typically are not the elements being searched. They can be a series of decisions or some operation arrangement. If there are multiple solutions, sometimes, people want to find the optimized one. For some spacial cases, there exist simplified approach such as the greedy methods. And dynamic programming can be used for more wide range of problems when they shows optimal substructures.

Bibliography

[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] M. Blum, R.W. Floyd, V. Pratt, R. Rivest and R. Tarjan, ”Time bounds for selection,” J. Comput. System Sci. 7 (1973) 448-461.

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

[5] Richard Bird. “Pearls of functional algorithm design”. Chapter 3. Cambridge University Press. 2010. ISBN, 1139490605, 9781139490603

[6] Edsger W. Dijkstra. “The saddleback search”. EWD-934. 1985. http://www.cs.utexas.edu/users/EWD/index09xx.html.

[7] Robert Boyer, and Strother Moore. “MJRTY - A Fast Majority Vote Algorithm”. Automated Reasoning: Essays in Honor of Woody Bledsoe, Automated Reasoning Series, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991, pp. 105-117.

[8] Cormode, Graham; S. Muthukrishnan (2004). “An Improved Data Stream Summary: The Count-Min Sketch and its Applications”. J. Algorithms 55: 29C38.

[9] Knuth Donald, Morris James H., jr, Pratt Vaughan. “Fast pattern matching in strings”. SIAM Journal on Computing 6 (2): 323C350. 1977.

[10] Robert Boyer, Strother Moore. “A Fast String Searching Algorithm”. Comm. ACM (New York, NY, USA: Association for Computing Machinery) 20 (10): 762C772. 1977

[11] R. N. Horspool. “Practical fast searching in strings”. Software - Practice & Experience 10 (6): 501C506. 1980.

[12] Wikipedia. “Boyer-Moore string search algorithm”. http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm

[13] Wikipedia. “Eight queens puzzle”. http://en.wikipedia.org/wiki/Eight_queens_puzzle

[14] George Pólya. "How to solve it: A new aspect of mathematical method". Princeton University Press(April 25, 2004). ISBN-13: 978-0691119663

[15] Wikipedia. "David A. Huffman". http://en.wikipedia.org/wiki/David_A._Huffman

[16] Fethi Rabhi, Guy Lapalme "Algorithms: a functional programming approach". Second edition. Addison-Wesley.