From grape to the world cup, the evolution of selection sort

Liu Xinyu [*]

February 10, 2013

1 Introduction

We have introduced the 'hello world' sorting algorithm, insertion sort. In this short chapter, we explain another straightforward sorting method, selection sort. The basic version of selection sort doesn't perform as good as the divide and conqueror methods, e.g. quick sort and merge sort. We'll use the same approaches in the chapter of insertion sort, to analyze why it's slow, and try to improve it by varies of attempts till reach the best bound of comparison based sorting, $O(N \lg N)$, by evolving to heap sort.

The idea of selection sort can be illustrated by a real life story. Consider a kid eating a bunch of grapes. There are two types of children according to my observation. One is optimistic type, that the kid always eats the biggest grape he/she can ever find; the other is pessimistic, that he/she always eats the smallest one.

The first type of kids actually eat the grape in an order that the size decreases monotonically; while the other eat in a increase order. The kid *sorts* the grapes in order of size in fact, and the method used here is selection sort.

Based on this idea, the algorithm of selection sort can be directly described as the following.

In order to sort a series of elements:

- The trivial case, if the series is empty, then we are done, the result is also empty;
- Otherwise, we find the smallest element, and append it to the tail of the result;

Note that this algorithm sorts the elements in increase order; It's easy to sort in decrease order by picking the biggest element instead; We'll introduce about passing a comparator as a parameter later on.

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

Figure 1: Always picking the smallest grape.

This description can be formalized to a equation.

$$sort(A) = \begin{cases} \Phi & : A = \Phi \\ \{m\} \cup sort(A') & : otherwise \end{cases} \tag{1}$$

Where m is the minimum element among collection A, and A' is all the rest elements except m:

$$\begin{array}{l} m = min(A) \\ A' = A - \{m\} \end{array}$$

We don't limit the data structure of the collection here. Typically, A is an array in imperative environment, and a list (singly linked-list particularly) in functional environment, and it can even be other data struture which will be introduced later.

The algorithm can also be given in imperative manner.

```
function SORT($A$)
    $X \leftarrow \Phi$
    while $A \neq \Phi$ do
        $x \leftarrow$ MIN($A$)
        $A \leftarrow$ DEL($A, x$)
        $X \leftarrow$ APPEND($X, x$)
    return $X$
```

Figure 2 depicts the process of this algorithm.

We just translate the very original idea of 'eating grapes' line by line without considering any expense of time and space. This realization stores the result in X, and when an selected element is appended to X, we delete the same element from A. This indicates that we can change it to 'in-place' sorting to reuse the spaces in A.

The idea is to store the minimum element in the first cell in A (we use term 'cell' if A is an array, and say 'node' if A is a list); then store the second

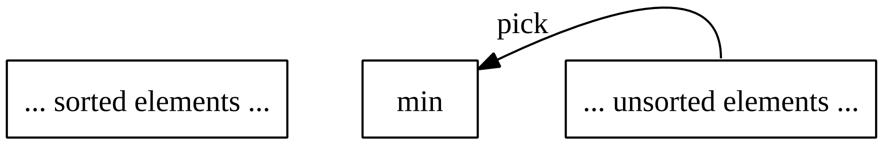

Figure 2: The left part is sorted data, continuously pick the minimum element in the rest and append it to the result.

minimum element in the next cell, then the third cell, ...

One solution to realize this sorting strategy is swapping. When we select the i-th minimum element, we swap it with the element in the i-th cell:

```
function SORT($A$)
    for $i \leftarrow 1$ to $|A|$ do
        $m \leftarrow$ MIN($A[i...]$)
        EXCHANGE $A[i] \leftrightarrow m$
```

Denote $A = \{a_1, a_2, ..., a_N\}$. At any time, when we process the i-th element, all elements before i, as $\{a_1, a_2, ..., a_{i-1}\}$ have already been sorted. We locate the minimum element among the $\{a_i, a_{i+1}, ..., a_N\}$, and exchange it with a_i, so that the i-th cell contains the right value. The process is repeatedly executed until we arrived at the last element.

This idea can be illustrated by figure 3.

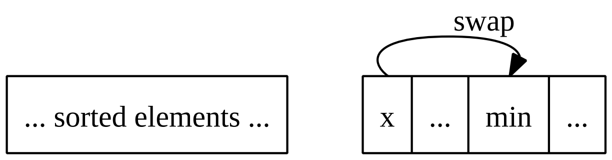

Figure 3: The left part is sorted data, continuously pick the minimum element in the rest and put it to the right position.

2 Finding the minimum

We haven't completely realized the selection sort, because we take the operation of finding the minimum (or the maximum) element as a black box. It's a puzzle how does a kid locate the biggest or the smallest grape. And this is an interesting topic for computer algorithms.

The easiest but not so fast way to find the minimum in a collection is to perform a scan. There are several ways to interpret this scan process. Consider that we want to pick the biggest grape. We start from any grape, compare it with another one, and pick the bigger one; then we take a next grape and compare it with the one we selected so far, pick the bigger one and go on the take-and-compare process, until there are not any grapes we haven't compared.

It's easy to get loss in real practice if we don't mark which grape has been compared. There are two ways to to solve this problem, which are suitable for different data-structures respectively.

2.1 Labeling

Method 1 is to label each grape with a number: $\{1, 2, ..., N\}$, and we systematically perform the comparison in the order of this sequence of labels. That we first compare grape number 1 and grape number 2, pick the bigger one; then we take grape number 3, and do the comparison, ... We repeat this process until arrive at grape number N. This is quite suitable for elements stored in an array.

```
function MIN($A$)
    $m \leftarrow A[1]$
    for $i \leftarrow 2$ to $|A|$ do
        if $A[i] < m$ then
            $m \leftarrow A[i]$
    return $m$
```

With MIN defined, we can complete the basic version of selection sort (or naive version without any optimization in terms of time and space).

However, this algorithm returns the value of the minimum element instead of its location (or the label of the grape), which needs a bit tweaking for the in-place version. Some languages such as ISO C++, support returning the reference as result, so that the swap can be achieved directly as below.

```
template<typename T>
T& min(T* from, T* to) {
    T* m;
    for (m = from++; from != to; ++from)
        if (*from < *m)
            m = from;
    return *m;
}

template<typename T>
void ssort(T* xs, int n) {
    int i;
    for (i = 0; i < n; ++i)
        std::swap(xs[i], min(xs+i, xs+n));
}
```

In environments without reference semantics, the solution is to return the location of the minimum element instead of the value:

```
function MIN-AT($A$)
    $m \leftarrow$ FIRST-INDEX($A$)
    for $i \leftarrow m + 1$ to $|A|$ do
```

```
        if $A[i] < A[m]$ then
            $m \leftarrow i$
    return $m$
```

Note that since we pass $A[i...]$ to Min-At as the argument, we assume the first element $A[i]$ as the smallest one, and examine all elements $A[i+1], A[i+2], ...$ one by one. Function First-Index() is used to retrieve i from the input parameter.

The following Python example program, for example, completes the basic in-place selection sort algorithm based on this idea. It explicitly passes the range information to the function of finding the minimum location.

```
def ssort(xs):
    n = len(xs)
    for i in range(n):
        m = min_at(xs, i, n)
        (xs[i], xs[m]) = (xs[m], xs[i])
    return xs

def min_at(xs, i, n):
    m = i;
    for j in range(i+1, n):
        if xs[j] < xs[m]:
            m = j
    return m
```

2.2 Grouping

Another method is to group all grapes in two parts: the group we have examined, and the rest we haven't. We denote these two groups as A and B; All the elements (grapes) as L. At the beginning, we haven't examine any grapes at all, thus A is empty (Φ), and B contains all grapes. We can select arbitrary two grapes from B, compare them, and put the loser (the smaller one for example) to A. After that, we repeat this process by continuously picking arbitrary grapes from B, and compare with the winner of the previous time until B becomes empty. At this time being, the final winner is the minimum element. And A turns to be $L - \{min(L)\}$, which can be used for the next time minimum finding.

There is an invariant of this method, that at any time, we have $L = A \cup \{m\} \cup B$, where m is the winner so far we hold.

This approach doesn't need the collection of grapes being indexed (as being labeled in method 1). It's suitable for any traversable data structures, including linked-list etc. Suppose b_1 is an arbitrary element in B if B isn't empty, and B' is the rest of elements with b_1 being removed, this method can be formalized as the below auxiliary function.

$$min'(A, m, B) = \begin{cases} (m, A) & : \quad B = \Phi \\ min'(A \cup \{m\}, b_1, B') & : \quad b_1 < m \\ min'(A \cup \{b_1\}, m, B') & : \quad otherwise \end{cases} \tag{2}$$

In order to pick the minimum element, we call this auxiliary function by passing an empty A, and use an arbitrary element (for instance, the first one) to initialize m:

$$extractMin(L) = min'(\Phi, l_1, L') \tag{3}$$

Where L' is all elements in L except for the first one l_1. The algorithm *extractMin*) doesn't not only find the minimum element, but also returns the updated collection which doesn't contain this minimum. Summarize this minimum extracting algorithm up to the basic selection sort definition, we can create a complete functional sorting program, for example as this Haskell code snippet.

```
sort [] = []
sort xs = x : sort xs' where
  (x, xs') = extractMin xs

extractMin (x:xs) = min' [] x xs where
  min' ys m [] = (m, ys)
  min' ys m (x:xs) = if m < x then min' (x:ys) m xs else min' (m:ys) x xs
```

The first line handles the trivial edge case that the sorting result for empty list is obvious empty; The second clause ensures that, there is at least one element, that's why the `extractMin` function needn't other pattern-matching.

One may think the second clause of `min'` function should be written like below:

```
min' ys m (x:xs) = if m < x then min' ys ++ [x] m xs
                             else min' ys ++ [m] x xs
```

Or it will produce the updated list in reverse order. Actually, it's necessary to use 'cons' instead of appending here. This is because appending is linear operation which is proportion to the length of part A, while 'cons' is constant $O(1)$ time operation. In fact, we needn't keep the relative order of the list to be sorted, as it will be re-arranged anyway during sorting.

It's quite possible to keep the relative order during sorting, while ensure the performance of finding the minimum element not degrade to quadratic. The following equation defines a solution.

$$extractMin(L) = \begin{cases} (l_1, \Phi) & : \quad |L| = 1 \\ (l_1, L') & : \quad l_1 < m, (m, L'') = extractMin(L') \\ (m, l_1 \cup L'') & : \quad otherwise \end{cases} \tag{4}$$

If L is a singleton, the minimum is the only element it contains. Otherwise, denote l_1 as the first element in L, and L' contains the rest elements except for

l_1, that $L' = \{l_2, l_3, ...\}$. The algorithm recursively finding the minimum element in L', which yields the intermediate result as (m, L''), that m is the minimum element in L', and L'' contains all rest elements except for m. Comparing l_1 with m, we can determine which of them is the final minimum result.

The following Haskell program implements this version of selection sort.

```
sort [] = []
sort xs = x : sort xs' where
  (x, xs') = extractMin xs

extractMin [x] = (x, [])
extractMin (x:xs) = if x < m then (x, xs) else (m, x:xs') where
  (m, xs') = extractMin xs
```

Note that only 'cons' operation is used, we needn't appending at all because the algorithm actually examines the list from right to left. However, it's not free, as this program need book-keeping the context (via call stack typically). The relative order is ensured by the nature of recursion. Please refer to the appendix about tail recursion call for detailed discussion.

2.3 performance of the basic selection sorting

Both the labeling method, and the grouping method need examine all the elements to pick the minimum in every round; and we totally pick up the minimum element N times. Thus the performance is around $N + (N-1) + (N-2) + ... + 1$ which is $\frac{N(N+1)}{2}$. Selection sort is a quadratic algorithm bound to $O(N^2)$ time.

Compare to the insertion sort, which we introduced previously, selection sort performs same in its best case, worst case and average case. While insertion sort performs well in best case (that the list has been reverse ordered, and it is stored in linked-list) as $O(N)$, and the worst performance is $O(N^2)$.

In the next sections, we'll examine, why selection sort performs poor, and try to improve it step by step.

Exercise 1

- Implement the basic imperative selection sort algorithm (the none in-place version) in your favorite programming language. Compare it with the in-place version, and analyze the time and space effectiveness.

3 Minor Improvement

3.1 Parameterize the comparator

Before any improvement in terms of performance, let's make the selection sort algorithm general enough to handle different sorting criteria.

We've seen two opposite examples so far, that one may need sort the elements in ascending order or descending order. For the former case, we need repeatedly finding the minimum, while for the later, we need find the maximum instead. They are just two special cases. In real world practice, one may want to sort things in varies criteria, e.g. in terms of size, weight, age, ...

One solution to handle them all is to passing the criteria as a compare function to the basic selection sort algorithms. For example:

$$sort(c, L) = \begin{cases} \Phi & : L = \Phi \\ m \cup sort(c, L'') & : otherwise, (m, L'') = extract(c, L') \end{cases} \quad (5)$$

And the algorithm $extract(c, L)$ is defined as below.

$$extract(c, L) = \begin{cases} (l_1, \Phi) & : |L| = 1 \\ (l_1, L') & : c(l_1, m), (m, L'') = extract(c, L') \\ (m, \{l_1\} \cup L'') & : \neg c(l_1, m) \end{cases} \quad (6)$$

Where c is a comparator function, it takes two elements, compare them and returns the result of which one is preceding of the other. Passing 'less than' operator ($<$) turns this algorithm to be the version we introduced in previous section.

Some environments require to pass the total ordering comparator, which returns result among 'less than', 'equal', and 'greater than'. We needn't such strong condition here, that c only tests if 'less than' is satisfied. However, as the minimum requirement, the comparator should meet the strict weak ordering as following [3]:

- Irreflexivity, for all x, it's not the case that $x < x$;
- Asymmetric, For all x and y, if $x < y$, then it's not the case $y < x$;
- Transitivity, For all x, y, and z, if $x < y$, and $y < z$, then $x < z$;

The following Scheme/Lisp program translates this generic selection sorting algorithm. The reason why we choose Scheme/Lisp here is because the lexical scope can simplify the needs to pass the 'less than' comparator for every function calls.

```
(define (sel-sort-by ltp? lst)
  (define (ssort lst)
    (if (null? lst)
        lst
        (let ((p (extract-min lst)))
          (cons (car p) (ssort (cdr p))))))
  (define (extract-min lst)
    (if (null? (cdr lst))
        lst
```

```
      (let ((p (extract-min (cdr lst))))
        (if (ltp? (car lst) (car p))
            lst
            (cons (car p) (cons (car lst) (cdr p)))))))
  (ssort lst))
```

Note that, both `ssort` and `extract-min` are inner functions, so that the 'less than' comparator `ltp?` is available to them. Passing '$<$' to this function yields the normal sorting in ascending order:

```
(sel-sort-by < '(3 1 2 4 5 10 9))
;Value 16: (1 2 3 4 5 9 10)
```

It's possible to pass varies of comparator to imperative selection sort as well. This is left as an exercise to the reader.

For the sake of brevity, we only consider sorting elements in ascending order in the rest of this chapter. And we'll not pass comparator as a parameter unless it's necessary.

3.2 Trivial fine tune

The basic in-place imperative selection sorting algorithm iterates all elements, and picking the minimum by traversing as well. It can be written in a compact way, that we inline the minimum finding part as an inner loop.

```
procedure SORT($A$)
    for $i \leftarrow 1$ to $|A|$ do
        $m \leftarrow i$
        for $j \leftarrow i + 1$ to $|A|$ do
            if $A[i] < A[m]$ then
                $m \leftarrow i$
        EXCHANGE $A[i] \leftrightarrow A[m]$
```

Observe that, when we are sorting N elements, after the first $N-1$ minimum ones are selected, the left only one, is definitely the N-th big element, so that we need NOT find the minimum if there is only one element in the list. This indicates that the outer loop can iterate to $N-1$ instead of N.

Another place we can fine tune, is that we needn't swap the elements if the i-th minimum one is just $A[i]$. The algorithm can be modified accordingly as below:

```
procedure SORT($A$)
    for $i \leftarrow 1$ to $|A| - 1$ do
        $m \leftarrow i$
        for $j \leftarrow i + 1$ to $|A|$ do
            if $A[i] < A[m]$ then
                $m \leftarrow i$
        if $m \neq i$ then
            EXCHANGE $A[i] \leftrightarrow A[m]$
```

Definitely, these modifications won't affects the performance in terms of big-O.

3.3 Cock-tail sort

Knuth gave an alternative realization of selection sort in [1]. Instead of selecting the minimum each time, we can select the maximum element, and put it to the last position. This method can be illustrated by the following algorithm.

```
procedure Sort'($A$)
    for $i \leftarrow |A|$ down-to 2 do
        $m \leftarrow i$
        for $j \leftarrow 1$ to $i - 1$ do
            if $A[m] < A[i]$ then
                $m \leftarrow i$
        Exchange $A[i] \leftrightarrow A[m]$
```

As shown in figure 4, at any time, the elements on right most side are sorted. The algorithm scans all unsorted ones, and locate the maximum. Then, put it to the tail of the unsorted range by swapping.

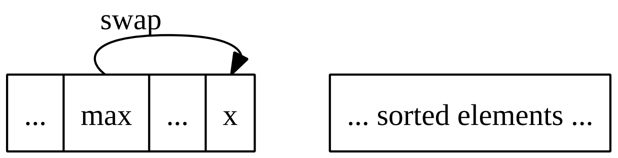

Figure 4: Select the maximum every time and put it to the end.

This version reveals the fact that, selecting the maximum element can sort the element in ascending order as well. What's more, we can find both the minimum and the maximum elements in one pass of traversing, putting the minimum at the first location, while putting the maximum at the last position. This approach can speed up the sorting slightly (halve the times of the outer loop).

```
procedure Sort($A$)
    for $i \leftarrow 1$ to $\lfloor \frac{|A|}{2} \rfloor$ do
        $min \leftarrow i$
        $max \leftarrow |A| + 1 - i$
        if $A[max] < A[min]$ then
            Exchange $A[min] \leftrightarrow A[max]$
        for $j \leftarrow i + 1$ to $|A| - i$ do
            if $A[j] < A[min]$ then
                $min \leftarrow j$
            if $A[max] < A[j]$ then
                $max \leftarrow j$
        EXCHANGE $A[i] \leftrightarrow A[min]$
        EXCHANGE $A[|A| + 1 - i] \leftrightarrow A[max]$
```

This algorithm can be illustrated as in figure 5, at any time, the left most and right most parts contain sorted elements so far. That the smaller sorted ones are on the left, while the bigger sorted ones are on the right. The algorithm scans the unsorted ranges, located both the minimum and the maximum positions, then put them to the head and the tail position of the unsorted ranges by swapping.

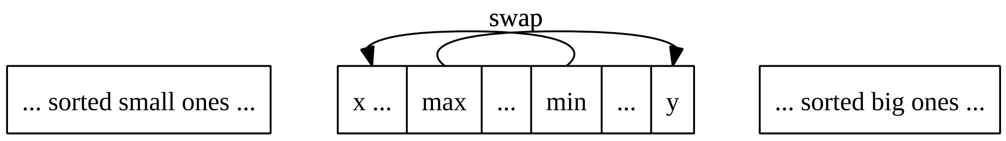

Figure 5: Select both the minimum and maximum in one pass, and put them to the proper positions.

Note that it's necessary to swap the left most and right most elements before the inner loop if they are not in correct order. This is because we scan the range excluding these two elements. Another method is to initialize the first element of the unsorted range as both the maximum and minimum before the inner loop. However, since we need two swapping operations after the scan, it's possible that the first swapping moves the maximum or the minimum from the position we just found, which leads the second swapping malfunctioned. How to solve this problem is left as exercise to the reader.

The following Python example program implements this cock-tail sort algorithm.

```
def cocktail_sort(xs):
    n = len(xs)
    for i in range(n / 2):
        (mi, ma) = (i, n - 1 -i)
        if xs[ma] < xs[mi]:
            (xs[mi], xs[ma]) = (xs[ma], xs[mi])
        for j in range(i+1, n - 1 - i):
            if xs[j] < xs[mi]:
                mi = j
            if xs[ma] < xs[j]:
                ma = j
        (xs[i], xs[mi]) = (xs[mi], xs[i])
        (xs[n - 1 - i], xs[ma]) = (xs[ma], xs[n - 1 - i])
    return xs
```

It's possible to realize cock-tail sort in functional approach as well. An intuitive recursive description can be given like this:

- Trivial edge case: If the list is empty, or there is only one element in the list, the sorted result is obviously the origin list;

- Otherwise, we select the minimum and the maximum, put them in the head and tail positions, then recursively sort the rest elements.

This algorithm description can be formalized by the following equation.

$$sort(L) = \left\{ \begin{array}{rcl} L & : & |L| \leq 1 \\ \{l_{min}\} \cup sort(L'') \cup \{l_{max}\} & : & otherwise \end{array} \right. \tag{7}$$

Where the minimum and the maximum are extracted from L by a function $select(L)$.

$$(l_{min}, L'', l_{max}) = select(L)$$

Note that, the minimum is actually linked to the front of the recursive sort result. Its semantic is a constant $O(1)$ time 'cons' (refer to the appendix of this book for detail). While the maximum is appending to the tail. This is typically a linear $O(N)$ time expensive operation. We'll optimize it later.

Function $select(L)$ scans the whole list to find both the minimum and the maximum. It can be defined as below:

$$select(L) = \left\{ \begin{array}{rcl} (min(l_1, l_2), max(l_1, l_2)) & : & L = \{l_1, l_2\} \\ (l_1, \{l_{min}\} \cup L'', l_{max}) & : & l_1 < l_{min} \\ (l_{min}, \{l_{max}\} \cup L'', l_1) & : & l_{max} < l_1 \\ (l_{min}, \{l_1\} \cup L'', l_{max}) & : & otherwise \end{array} \right. \tag{8}$$

Where $(l_{min}, L'', l_{max}) = select(L')$ and L' is the rest of the list except for the first element l_1. If there are only two elements in the list, we pick the smaller as the minimum, and the bigger as the maximum. After extract them, the list becomes empty. This is the trivial edge case; Otherwise, we take the first element l_1 out, then recursively perform selection on the rest of the list. After that, we compare if l_1 is less then the minimum or greater than the maximum candidates, so that we can finalize the result.

Note that for all the cases, there is no appending operation to form the result. However, since selection must scan all the element to determine the minimum and the maximum, it is bound to $O(N)$ linear time.

The complete example Haskell program is given as the following.

```haskell
csort [] = []
csort [x] = [x]
csort xs = mi : csort xs' ++ [ma] where
  (mi, xs', ma) = extractMinMax xs

extractMinMax [x, y] = (min x y, [], max x y)
extractMinMax (x:xs) | x < mi = (x, mi:xs', ma)
                     | ma < x = (mi, ma:xs', x)
                     | otherwise = (mi, x:xs', ma)
  where (mi, xs', ma) = extractMinMax xs
```

We mentioned that the appending operation is expensive in this intuitive version. It can be improved. This can be achieved in two steps. The first step is to convert the cock-tail sort into tail-recursive call. Denote the sorted small ones as A, and sorted big ones as B in figure 5. We use A and B as accumulators. The new cock-tail sort is defined as the following.

$$sort'(A, L, B) = \begin{cases} A \cup L \cup B & : & L = \Phi \lor |L| = 1 \\ sort'(A \cup \{l_{min}\}, L'', \{l_{max}\} \cup B) & : & otherwise \end{cases} \tag{9}$$

Where l_{min}, l_{max} and L'' are defined as same as before. And we start sorting by passing empty A and B: $sort(L) = sort'(\Phi, L, \Phi)$.

Besides the edge case, observing that the appending operation only happens on $A \cup \{l_{min}\}$; while l_{max} is only linked to the head of B. This appending occurs in every recursive call. To eliminate it, we can store A in reverse order as $\overleftarrow{A}$, so that l_{max} can be 'cons' to the head instead of appending. Denote $cons(x, L) = \{x\} \cup L$ and $append(L, x) = L \cup \{x\}$, we have the below equation.

$$\begin{array}{rl} append(L, x) & = reverse(cons(x, reverse(L))) \\ & = reverse(cons(x, \overleftarrow{L})) \end{array} \tag{10}$$

Finally, we perform a reverse to turn $\overleftarrow{A}$ back to A. Based on this idea, the algorithm can be improved one more step as the following.

$$sort'(A, L, B) = \begin{cases} reverse(A) \cup B & : & L = \Phi \\ reverse(\{l_1\} \cup A) \cup B & : & |L| = 1 \\ sort'(\{l_{min}\} \cup A, L'', \{l_{max}\} \cup B) & : & \end{cases} \tag{11}$$

This algorithm can be implemented by Haskell as below.

```
csort' xs = cocktail [] xs [] where
  cocktail as [] bs = reverse as ++ bs
  cocktail as [x] bs = reverse (x:as) ++ bs
  cocktail as xs bs = let (mi, xs', ma) = extractMinMax xs
                      in cocktail (mi:as) xs' (ma:bs)
```

Exercise 2

- Realize the imperative basic selection sort algorithm, which can take a comparator as a parameter. Please try both dynamic typed language and static typed language. How to annotate the type of the comparator as general as possible in a static typed language?
- Implement Knuth's version of selection sort in your favorite programming language.
- An alternative to realize cock-tail sort is to assume the i-th element both the minimum and the maximum, after the inner loop, the minimum and maximum are found, then we can swap the the minimum to the i-th

position, and the maximum to position $|A|+1-i$. Implement this solution in your favorite imperative language. Please note that there are several special edge cases should be handled correctly:

- $A = \{max, min, ...\}$;
- $A = \{..., max, min\}$;
- $A = \{max, ..., min\}$.

Please don't refer to the example source code along with this chapter before you try to solve this problem.

4 Major improvement

Although cock-tail sort halves the numbers of loop, the performance is still bound to quadratic time. It means that, the method we developed so far handles big data poorly compare to other divide and conquer sorting solutions.

To improve selection based sort essentially, we must analyze where is the bottle-neck. In order to sort the elements by comparison, we must examine all the elements for ordering. Thus the outer loop of selection sort is necessary. However, must it scan all the elements every time to select the minimum? Note that when we pick the smallest one at the first time, we actually traverse the whole collection, so that we know which ones are relative big, and which ones are relative small partially.

The problem is that, when we select the further minimum elements, instead of re-using the ordering information we obtained previously, we drop them all, and blindly start a new traverse.

So the key point to improve selection based sort is to re-use the previous result. There are several approaches, we'll adopt an intuitive idea inspired by football match in this chapter.

4.1 Tournament knock out

The football world cup is held every four years. There are 32 teams from different continent play the final games. Before 1982, there were 16 teams compete for the tournament finals[4].

For simplification purpose, let's go back to 1978 and imagine a way to determine the champion: In the first round, the teams are grouped into 8 pairs to play the game; After that, there will be 8 winner, and 8 teams will be out. Then in the second round, these 8 teams are grouped into 4 pairs. This time there will be 4 winners after the second round of games; Then the top 4 teams are divided into 2 pairs, so that there will be only two teams left for the final game.

The champion is determined after the total 4 rounds of games. And there are actually $8 + 4 + 2 + 1 = 16$ games. Now we have the world cup champion, however, the world cup game won't finish at this stage, we need to determine which is the silver medal team.

Readers may argue that isn't the team beaten by the champion at the final game the second best? This is true according to the real world cup rule. However, it isn't fair enough in some sense.

We often heard about the so called 'group of death', Let's suppose that Brazil team is grouped with Deutch team at the very beginning. Although both teams are quite strong, one of them must be knocked out. It's quite possible that even the team loss that game can beat all the other teams except for the champion. Figure 6 illustrates such case.

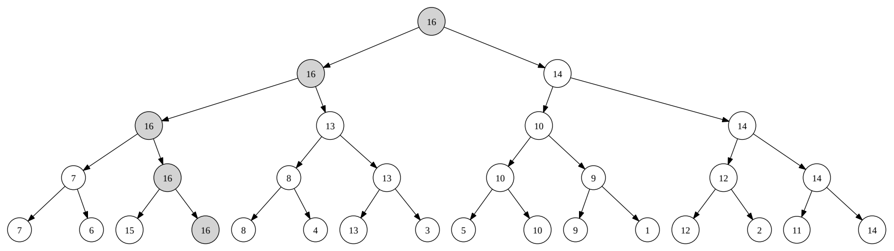

Figure 6: The element 15 is knocked out in the first round.

Imagine that every team has a number. The bigger the number, the stronger the team. Suppose that the stronger team always beats the team with smaller number, although this is not true in real world. But this simplification is fair enough for us to develop the tournament knock out solution. This maximum number which represents the champion is 16. Definitely, team with number 14 isn't the second best according to our rules. It should be 15, which is knocked out at the first round of comparison.

The key question here is to find an effective way to locate the second maximum number in this tournament tree. After that, what we need is to apply the same method to select the third, the fourth, ..., to accomplish the selection based sort.

One idea is to assign the champion a very small number (for instance, $-\infty$), so that it won't be selected next time, and the second best one, becomes the new champion. However, suppose there are 2^m teams for some natural number m, it still takes $2^{m-1} + 2^{m-2} + ... + 2 + 1 = 2^m$ times of comparison to determine the new champion. Which is as slow as the first time.

Actually, we needn't perform a bottom-up comparison at all since the tournament tree stores plenty of ordering information. Observe that, the second best team must be beaten by the champion at sometime, or it will be the final winner. So we can track the path from the root of the tournament tree to the leaf of the champion, examine all the teams along with this path to find the second best team.

In figure 6, this path is marked in gray color, the elements to be examined are $\{14, 13, 7, 15\}$. Based on this idea, we refine the algorithm like below.

1. Build a tournament tree from the elements to be sorted, so that the champion (the maximum) becomes the root;
2. Extract the root from the tree, perform a top-down pass and replace the maximum with $-\infty$;
3. Perform a bottom-up back-track along the path, determine the new champion and make it as the new root;
4. Repeat step 2 until all elements have been extracted.

Figure 7, 8, and 9 show the steps of applying this strategy.

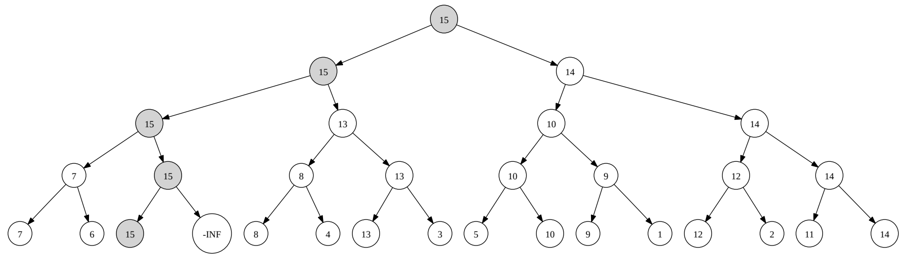

Figure 7: Extract 16, replace it with $-\infty$, 15 sifts up to root.

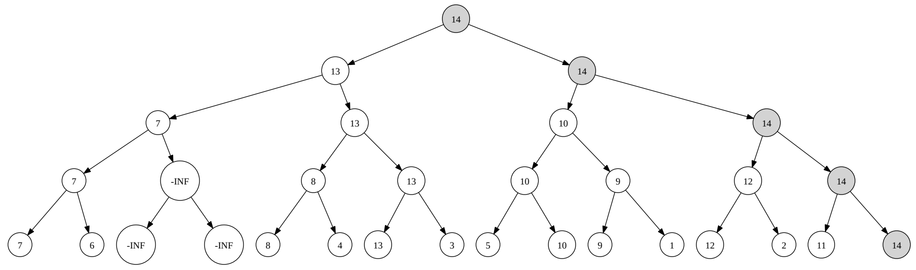

Figure 8: Extract 15, replace it with $-\infty$, 14 sifts up to root.

We can reuse the binary tree definition given in the first chapter of this book to represent tournament tree. In order to back-track from leaf to the root, every node should hold a reference to its parent (concept of pointer in some environment such as ANSI C):

```
struct Node {
  Key key;
  struct Node *left, *right, *parent;
};
```

To build a tournament tree from a list of elements (suppose the number of elements are 2^m for some m), we can first wrap each element as a leaf, so that we obtain a list of binary trees. We take every two trees from this list, compare

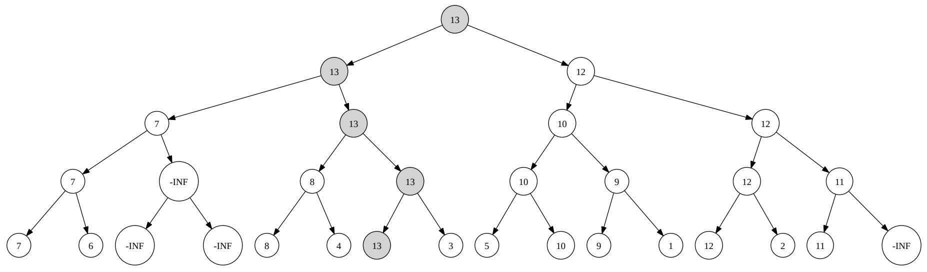

Figure 9: Extract 14, replace it with $-\infty$, 13 sifts up to root.

their keys, and form a new binary tree with the bigger key as the root; the two trees are set as the left and right children of this new binary tree. Repeat this operation to build a new list of trees. The height of each tree is increased by 1. Note that the size of the tree list halves after such a pass, so that we can keep reducing the list until there is only one tree left. And this tree is the finally built tournament tree.

```
function BUILD-TREE($A$)
    $T \leftarrow \Phi$
    for each $x \in A$ do
        $t \leftarrow$ CREATE-NODE
        KEY($t$) $\leftarrow x$
        APPEND($T, t$)
    while $|T| > 1$ do
        $T' \leftarrow \Phi$
        for every $t_1, t_2 \in$ T do
            $t \leftarrow$ CREATE-NODE
            KEY($t$) $\leftarrow$ MAX(KEY($t_1$), KEY($t_2$))
            LEFT($t$) $\leftarrow t_1$
            RIGHT($t$) $\leftarrow t_2$
            PARENT($t_1$) $\leftarrow t$
            PARENT($t_2$) $\leftarrow t$
            APPEND($T', t$)
        $T \leftarrow T'$
    return $T[1]$
```

Suppose the length of the list A is N, this algorithm firstly traverses the list to build tree, which is linear to N time. Then it repeatedly compares pairs, which loops proportion to $N + \frac{N}{2} + \frac{N}{4} + ... + 2 = 2N$. So the total performance is bound to $O(N)$ time.

The following ANSI C program implements this tournament tree building algorithm.

```
struct Node* build(const Key* xs, int n) {
    int i;
    struct Node *t, **ts = (struct Node**) malloc(sizeof(struct Node*) * n);
    for (i = 0; i < n; ++i)
        ts[i] = leaf(xs[i]);
    for (; n > 1; n /= 2)
        for (i = 0; i < n; i += 2)
            ts[i/2] = branch(max(ts[i]->key, ts[i+1]->key), ts[i], ts[i+1]);
    t = ts[0];
    free(ts);
    return t;
}
```

The type of key can be defined somewhere, for example:

```
typedef int Key;
```

Function `leaf(x)` creats a leaf node, with value `x` as key, and sets all its fields, left, right and parent to `NIL`. While function `branch(key, left, right)` creates a branch node, and links the new created node as parent of its two children if they are not empty. For the sake of brevity, we skip the detail of them. They are left as exercise to the reader, and the complete program can be downloaded along with this book.

Some programming environments, such as Python provides tool to iterate every two elements at a time, for example:

```
for x, y in zip(*[iter(ts)]*2):
```

We skip such language specific feature, readers can refer to the Python example program along with this book for details.

When the maximum element is extracted from the tournament tree, we replace it with $-\infty$, and repeatedly replace all these values from the root to the leaf. Next, we back-track to root through the parent field, and determine the new maximum element.

```
function EXTRACT-MAX($T$)
    $m \leftarrow$ KEY($T$)
    KEY($T$) $\leftarrow -\infty$
    while $\neg$ LEAF?($T$) do                          ▷ The top down pass
        if KEY(LEFT($T$)) $= m$ then
            $T \leftarrow$ LEFT($T$)
        else
            $T \leftarrow$ RIGHT($T$)
        KEY($T$) $\leftarrow -\infty$
    while PARENT($T$) $\neq \Phi$ do                     ▷ The bottom up pass
        $T \leftarrow$ PARENT($T$)
        KEY($T$) $\leftarrow$ MAX(KEY(LEFT($T$)), KEY(RIGHT($T$)))
    return $m$
```

This algorithm returns the extracted maximum element, and modifies the tournament tree in-place. Because we can't represent $-\infty$ in real program by limited length of word, one approach is to define a relative negative big number, which is less than all the elements in the tournament tree, for example, suppose all the elements are greater than -65535, we can define negative infinity as below:

```
#define N_INF -65535
```

We can implements this algorithm as the following ANSI C example program.

```
Key pop(struct Node* t) {
    Key x = t→key;
    t→key = N_INF;
    while (!isleaf(t)) {
        t = t→left→key == x ? t→left : t→right;
        t→key = N_INF;
    }
    while (t→parent) {
        t = t→parent;
        t→key = max(t→left→key, t→right→key);
    }
    return x;
}
```

The behavior of EXTRACT-MAX is quite similar to the pop operation for some data structures, such as queue, and heap, thus we name it as `pop` in this code snippet.

Algorithm EXTRACT-MAX process the tree in tow passes, one is top-down, then a bottom-up along the path that the 'champion team wins the world cup'. Because the tournament tree is well balanced, the length of this path, which is the height of the tree, is bound to $O(\lg N)$, where N is the number of the elements to be sorted (which are equal to the number of leaves). Thus the performance of this algorithm is $O(\lg N)$.

It's possible to realize the tournament knock out sort now. We build a tournament tree from the elements to be sorted, then continuously extract the maximum. If we want to sort in monotonically increase order, we put the first extracted one to the right most, then insert the further extracted elements one by one to left; Otherwise if we want to sort in decrease order, we can just append the extracted elements to the result. Below is the algorithm sorts elements in ascending order.

```
procedure SORT($A$)
    $T \leftarrow$ BUILD-TREE($A$)
    for $i \leftarrow |A|$ down to 1 do
        $A[i] \leftarrow$ EXTRACT-MAX($T$)
```

Translating it to ANSI C example program is straightforward.

```
void tsort(Key* xs, int n) {
    struct Node* t = build(xs, n);
    while(n)
        xs[--n] = pop(t);
    release(t);
}
```

This algorithm firstly takes $O(N)$ time to build the tournament tree, then performs N pops to select the maximum elements so far left in the tree. Since each pop operation is bound to $O(\lg N)$, thus the total performance of tournament knock out sorting is $O(N \lg N)$.

4.1.1 Refine the tournament knock out

It's possible to design the tournament knock out algorithm in purely functional approach. And we'll see that the two passes (first top-down replace the champion with $-\infty$, then bottom-up determine the new champion) in pop operation can be combined in recursive manner, so that we needn't the parent field any more. We can re-use the functional binary tree definition as the following example Haskell code.

```
data Tr a = Empty | Br (Tr a) a (Tr a)
```

Thus a binary tree is either empty or a branch node contains a key, a left sub tree and a right sub tree. Both children are again binary trees.

We've use hard coded big negative number to represents $-\infty$. However, this solution is ad-hoc, and it forces all elements to be sorted are greater than this pre-defined magic number. Some programming environments support algebraic type, so that we can define negative infinity explicitly. For instance, the below Haskell program setups the concept of infinity [1].

```
data Infinite a = NegInf | Only a | Inf deriving (Eq, Ord)
```

From now on, we switch back to use the $min()$ function to determine the winner, so that the tournament selects the minimum instead of the maximum as the champion.

Denote function $key(T)$ returns the key of the tree rooted at T. Function $wrap(x)$ wraps the element x into a leaf node. Function $tree(l, k, r)$ creates a branch node, with k as the key, l and r as the two children respectively.

The knock out process, can be represented as comparing two trees, picking the smaller key as the new key, and setting these two trees as children:

$$branch(T_1, T_2) = tree(T_1, min(key(T_1), key(T_2)), T_2) \tag{12}$$

This can be implemented in Haskell word by word:

```
branch t1 t2 = Br t1 (min (key t1) (key t2)) t2
```

There is limitation in our tournament sorting algorithm so far. It only accepts collection of elements with size of 2^m, or we can't build a complete binary tree. This can be actually solved in the tree building process. Remind that we pick two trees every time, compare and pick the winner. This is perfect if there are always even number of trees. Considering a case in football match, that one team is absent for some reason (sever flight delay or whatever), so that there left one team without a challenger. One option is to make this team the

[1] The order of the definition of 'NegInf', regular number, and 'Inf' is significant if we want to derive the default, correct comparing behavior of 'Ord'. Anyway, it's possible to specify the detailed order by make it as an instance of 'Ord'. However, this is Language specific feature which is out of the scope of this book. Please refer to other textbook about Haskell.

winner, so that it will attend the further games. Actually, we can use the similar approach.

To build the tournament tree from a list of elements, we wrap every element into a leaf, then start the building process.

$$build(L) = build'(\{wrap(x) | x \in L\}) \tag{13}$$

The $build'(\mathbb{T})$ function terminates when there is only one tree left in $\mathbb{T}$, which is the champion. This is the trivial edge case. Otherwise, it groups every two trees in a pair to determine the winners. When there are odd numbers of trees, it just makes the last tree as the winner to attend the next level of tournament and recursively repeats the building process.

$$build'(\mathbb{T}) = \left\{ \begin{array}{rcl} \mathbb{T} & : & |\mathbb{T}| \leq 1 \\ build'(pair(\mathbb{T})) & : & otherwise \end{array} \right. \tag{14}$$

Note that this algorithm actually handles another special cases, that the list to be sort is empty. The result is obviously empty.

Denote $\mathbb{T} = \{T_1, T_2, ...\}$ if there are at least two trees, and $\mathbb{T}'$ represents the left trees by removing the first two. Function $pair(\mathbb{T})$ is defined as the following.

$$pair(\mathbb{T}) = \left\{ \begin{array}{rcl} \{branch(T_1, T_2)\} \cup pair(\mathbb{T}') & : & |\mathbb{T}| \geq 2 \\ \mathbb{T} & : & otherwise \end{array} \right. \tag{15}$$

The complete tournament tree building algorithm can be implemented as the below example Haskell program.

```
fromList :: (Ord a) ⇒ [a] → Tr (Infinite a)
fromList = build ∘ (map wrap) where
  build [] = Empty
  build [t] = t
  build ts = build $ pair ts
  pair (t1:t2:ts) = (branch t1 t2):pair ts
  pair ts = ts
```

When extracting the champion (the minimum) from the tournament tree, we need examine either the left child sub-tree or the right one has the same key, and recursively extract on that tree until arrive at the leaf node. Denote the left sub-tree of T as L, right sub-tree as R, and K as its key. We can define this popping algorithm as the following.

$$pop(T) = \left\{ \begin{array}{rcl} tree(\Phi, \infty, \Phi) & : & L = \Phi \wedge R = \Phi \\ tree(L', min(key(L'), key(R)), R) & : & K = key(L), L' = pop(L) \\ tree(L, min(key(L), key(R')), R') & : & K = key(R), R' = pop(R) \end{array} \right. \tag{16}$$

It's straightforward to translate this algorithm into example Haskell code.

```
pop (Br Empty _ Empty) = Br Empty Inf Empty
pop (Br l k r) | k == key l = let l' = pop l in Br l' (min (key l') (key r)) r
               | k == key r = let r' = pop r in Br l (min (key l) (key r')) r'
```

Note that this algorithm only removes the current champion without returning it. So it's necessary to define a function to get the champion at the root node.

$$top(T) = key(T) \tag{17}$$

With these functions defined, tournament knock out sorting can be formalized by using them.

$$sort(L) = sort'(build(L)) \tag{18}$$

Where $sort'(T)$ continuously pops the minimum element to form a result list

$$sort'(T) = \begin{cases} \Phi & : T = \Phi \lor key(T) = \infty \\ \{top(T)\} \cup sort'(pop(T)) & : otherwise \end{cases} \tag{19}$$

The rest of the Haskell code is given below to complete the implementation.

```
top = only ∘ key

tsort :: (Ord a) ⇒ [a] → [a]
tsort = sort' ∘ fromList where
    sort' Empty = []
    sort' (Br _ Inf _) = []
    sort' t = (top t) : (sort' $ pop t)
```

And the auxiliary function `only`, `key`, `wrap` accomplished with explicit infinity support are list as the following.

```
only (Only x) = x
key (Br _ k _ ) = k
wrap x = Br Empty (Only x) Empty
```

Exercise 3

- Implement the helper function `leaf()`, `branch`, `max()` `lsleaf()`, and `release()` to complete the imperative tournament tree program.
- Implement the imperative tournament tree in a programming language support GC (garbage collection).
- Why can our tournament tree knock out sort algorithm handle duplicated elements (elements with same value)? We say a sorting algorithm stable, if it keeps the original order of elements with same value. Is the tournament tree knock out sorting stable?
- Design an imperative tournament tree knock out sort algorithm, which satisfies the following:
    - Can handle arbitrary number of elements;

  - Without using hard coded negative infinity, so that it can take elements with any value.

- Compare the tournament tree knock out sort algorithm and binary tree sort algorithm, analyze efficiency both in time and space.
- Compare the heap sort algorithm and binary tree sort algorithm, and do same analysis for them.

4.2 Final improvement by using heap sort

We manage improving the performance of selection based sorting to $O(N \lg N)$ by using tournament knock out. This is the limit of comparison based sort according to [1]. However, there are still rooms for improvement. After sorting, there lefts a complete binary tree with all leaves and branches hold useless infinite values. This isn't space efficient at all. Can we release the nodes when popping?

Another observation is that if there are N elements to be sorted, we actually allocate about $2N$ tree nodes. N for leaves and N for branches. Is there any better way to halve the space usage?

The final sorting structure described in equation 19 can be easily uniformed to a more general one if we treat the case that the tree is empty if its root holds infinity as key:

$$sort'(T) = \begin{cases} \Phi & : T = \Phi \\ \{top(T)\} \cup sort'(pop(T)) & : otherwise \end{cases} \tag{20}$$

This is exactly as same as the one of heap sort we gave in previous chapter. Heap always keeps the minimum (or the maximum) on the top, and provides fast pop operation. The binary heap by implicit array encodes the tree structure in array index, so there aren't any extra spaces allocated except for the N array cells. The functional heaps, such as leftist heap and splay heap allocate N nodes as well. We'll introduce more heaps in next chapter which perform well in many aspects.

5 Short summary

In this chapter, we present the evolution process of selection based sort. selection sort is easy and commonly used as example to teach students about embedded looping. It has simple and straightforward structure, but the performance is quadratic. In this chapter, we do see that there exists ways to improve it not only by some fine tuning, but also fundamentally change the data structure, which leads to tournament knock out and heap sort.

References

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

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

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

[4] Wikipedia. “FIFA world cup”. http://en.wikipedia.org/wiki/FIFA_World_Cup