18 Selection Sort and Heapsort

Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

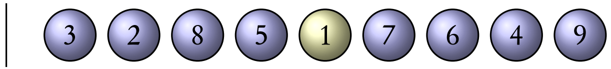
April 8, 2019

Outline

Selection Sort

Selection sort is to select the minimum element by traversal from the remaining elements and swap it with the element at the least unsorted index.

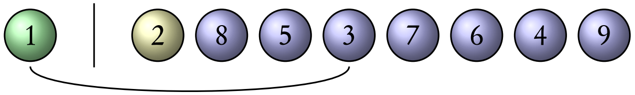
- The list is again divided into two parts, the front contains the sorted elements, the rear contains the unsorted elements.
- The unsorted part is treated as a priority queue, from which we select the minimum elements one by one.
- We alway append the selected elements to the end of the sorted part, because the selected elements are in ascending order.
- Since the priority queue is implemented as an unsorted list, we can put the original elements, whose places are taken, to the arbitrary places left by the selected elements.



Selection Sort

Selection sort is to select the minimum element by traversal from the remaining elements and swap it with the element at the least unsorted index.

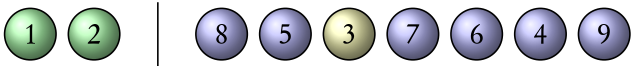
- The list is again divided into two parts, the front contains the sorted elements, the rear contains the unsorted elements.
- The unsorted part is treated as a priority queue, from which we select the minimum elements one by one.
- We alway append the selected elements to the end of the sorted part, because the selected elements are in ascending order.
- Since the priority queue is implemented as an unsorted list, we can put the original elements, whose places are taken, to the arbitrary places left by the selected elements.



Selection Sort

Selection sort is to select the minimum element by traversal from the remaining elements and swap it with the element at the least unsorted index.

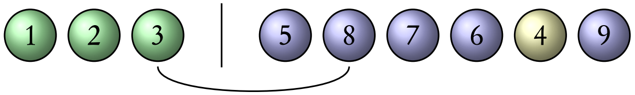
- The list is again divided into two parts, the front contains the sorted elements, the rear contains the unsorted elements.
- The unsorted part is treated as a priority queue, from which we select the minimum elements one by one.
- We alway append the selected elements to the end of the sorted part, because the selected elements are in ascending order.
- Since the priority queue is implemented as an unsorted list, we can put the original elements, whose places are taken, to the arbitrary places left by the selected elements.



Selection Sort

Selection sort is to select the minimum element by traversal from the remaining elements and swap it with the element at the least unsorted index.

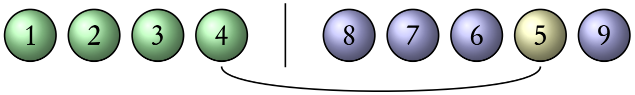
- The list is again divided into two parts, the front contains the sorted elements, the rear contains the unsorted elements.
- The unsorted part is treated as a priority queue, from which we select the minimum elements one by one.
- We alway append the selected elements to the end of the sorted part, because the selected elements are in ascending order.
- Since the priority queue is implemented as an unsorted list, we can put the original elements, whose places are taken, to the arbitrary places left by the selected elements.



Selection Sort

Selection sort is to select the minimum element by traversal from the remaining elements and swap it with the element at the least unsorted index.

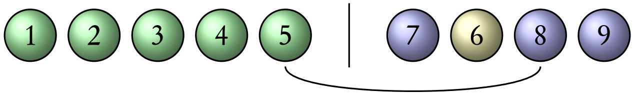
- The list is again divided into two parts, the front contains the sorted elements, the rear contains the unsorted elements.
- The unsorted part is treated as a priority queue, from which we select the minimum elements one by one.
- We alway append the selected elements to the end of the sorted part, because the selected elements are in ascending order.
- Since the priority queue is implemented as an unsorted list, we can put the original elements, whose places are taken, to the arbitrary places left by the selected elements.



Selection Sort

Selection sort is to select the minimum element by traversal from the remaining elements and swap it with the element at the least unsorted index.

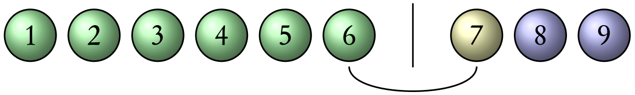
- The list is again divided into two parts, the front contains the sorted elements, the rear contains the unsorted elements.
- The unsorted part is treated as a priority queue, from which we select the minimum elements one by one.
- We alway append the selected elements to the end of the sorted part, because the selected elements are in ascending order.
- Since the priority queue is implemented as an unsorted list, we can put the original elements, whose places are taken, to the arbitrary places left by the selected elements.



Selection Sort

Selection sort is to select the minimum element by traversal from the remaining elements and swap it with the element at the least unsorted index.

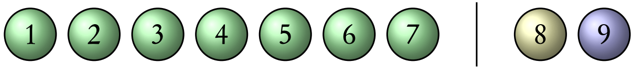
- The list is again divided into two parts, the front contains the sorted elements, the rear contains the unsorted elements.
- The unsorted part is treated as a priority queue, from which we select the minimum elements one by one.
- We alway append the selected elements to the end of the sorted part, because the selected elements are in ascending order.
- Since the priority queue is implemented as an unsorted list, we can put the original elements, whose places are taken, to the arbitrary places left by the selected elements.



Selection Sort

Selection sort is to select the minimum element by traversal from the remaining elements and swap it with the element at the least unsorted index.

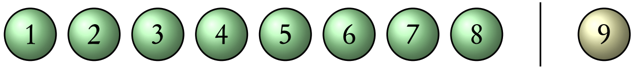
- The list is again divided into two parts, the front contains the sorted elements, the rear contains the unsorted elements.
- The unsorted part is treated as a priority queue, from which we select the minimum elements one by one.
- We alway append the selected elements to the end of the sorted part, because the selected elements are in ascending order.
- Since the priority queue is implemented as an unsorted list, we can put the original elements, whose places are taken, to the arbitrary places left by the selected elements.



Selection Sort

Selection sort is to select the minimum element by traversal from the remaining elements and swap it with the element at the least unsorted index.

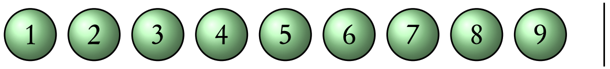
- The list is again divided into two parts, the front contains the sorted elements, the rear contains the unsorted elements.
- The unsorted part is treated as a priority queue, from which we select the minimum elements one by one.
- We alway append the selected elements to the end of the sorted part, because the selected elements are in ascending order.
- Since the priority queue is implemented as an unsorted list, we can put the original elements, whose places are taken, to the arbitrary places left by the selected elements.



Selection Sort

Selection sort is to select the minimum element by traversal from the remaining elements and swap it with the element at the least unsorted index.

- The list is again divided into two parts, the front contains the sorted elements, the rear contains the unsorted elements.
- The unsorted part is treated as a priority queue, from which we select the minimum elements one by one.
- We alway append the selected elements to the end of the sorted part, because the selected elements are in ascending order.
- Since the priority queue is implemented as an unsorted list, we can put the original elements, whose places are taken, to the arbitrary places left by the selected elements.



Selection Sort on Array-Based Lists

The straight selection scans all the remaining elements to find the minimum and swaps it with the element just next to the sorted ones. The function *selection_sort_a* sorts the elements of an array-based list *a*.

```
def selection_sort_a(a):
    n = len(a)
    for i in range(n-1):
        m = i
        for j in range(i+1, n):
            if not a[m] <= a[j]:
                m = j
        if i != m:
            a[i], a[m] = a[m], a[i]
```

Selection Sort on Singly Linked Lists

For a singly linked list pointed to by *h*, we initialize an empty list pointed to by *s*, then repeatedly select the *maximum* nodes (why?) from *h* and “push” them to the *front* of *s*. The function *selection_sort_l* sorts a singly linked list *h* and returns the head node of the sorted list.

```
def selection_sort_l(h):
    s = None
    while h is not None:
        m, r = h, None
        p, q = h.nxt, h
        while p is not None:
            if m.elm <= p.elm: m, r = p, q
            p, q = p.nxt, p
        if r is None: h = m.nxt
        else: r.nxt = m.nxt
        m.nxt, s = s, m
    return s
```

Analysis of Straight Selection Sort

For a list of n elements, we only need a fix amount of auxiliary space for selection sort.

- $\mathcal{O}(1)$ auxiliary space.

We count the number of element comparisons.

- In each inner loop, we must compare the first unsorted element with all the rest unsorted elements, thus there are

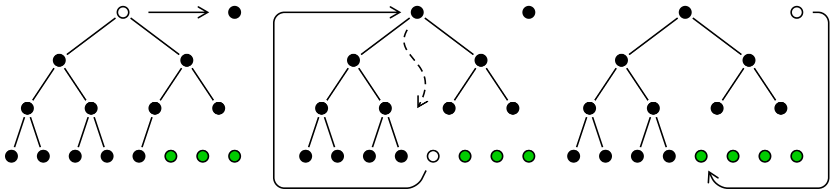
$$(n-1)+(n-2)+\cdots+1=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}$$

  comparisons, i.e. $\mathcal{O}(n^2)$.

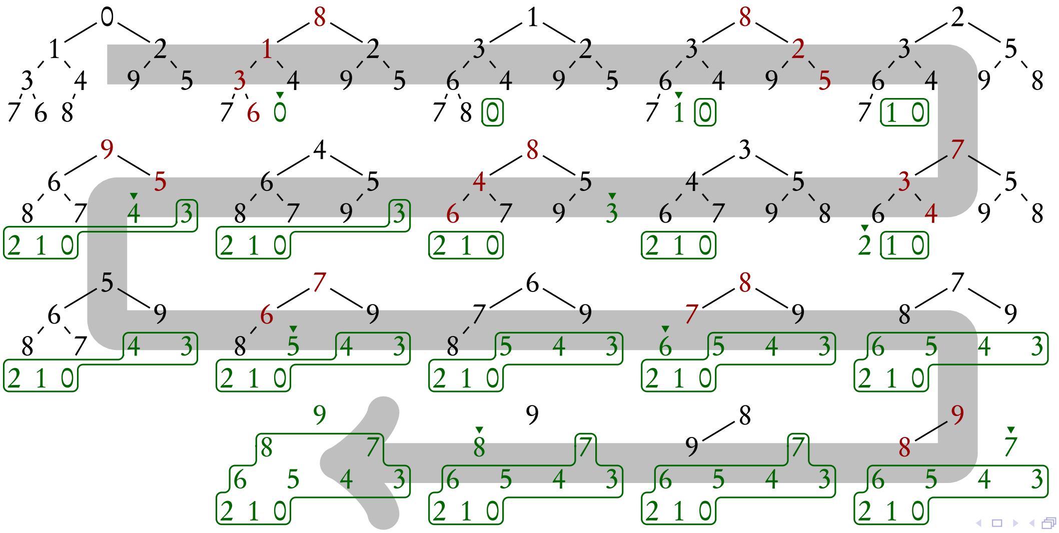
Selection sort is stable on linked lists. However, it is *unstable* on arrays for the algorithm given above, since in the unsorted portion, we swap a front element to the rear.

Selecting by a Heap

- We can also heapify the array-based list, and select the minimum elements by the heap. The array is divided into two parts — a heap and a subarray of sorted elements.
- The heap must occupy the front part of the array, since the tree relations must be computed according to the indices of the elements, and the indices must start from `0`.
- After a *pop_min*, the last element of the heap is moved to the root for being sifted down, leaving the last position of the heap empty.
- We have to put the extracted "min" elements to the empty positions at the end of the heap.
- The *reverse* order of the sorting must be used as the heap order.



Heapsort Illustrated



`

Analysis of Heapsort

For an array of n elements, we only need a fix amount of auxiliary space for heapsort.

- $\mathcal{O}(1)$ auxiliary space.

We count the number of element comparisons.

- It takes $\mathcal{O}(n)$ time to heapify an array.
- In the worst case, we sift the elements down the heap $n-1$ times to the bottom, as the heap is shrinking, the maximum depth for all the down-siftings is $\mathcal{O}(\log n)$, but half of the elements are at the bottom, and of depth $\mathcal{O}(\log n)$, so we have the upper bound $n \cdot \mathcal{O}(\log n)$ and the lower bound $\frac{n}{2} \cdot \mathcal{O}(\log n)$ of the time complexity of extracting elements from the heap.
- Heapsort takes overall $\mathcal{O}(n \log n)$ time to sort an array of size n.

Heapsort is *not* stable, since elements of the same key can stay in different sub-trees, be sifted up and down independently.

A General Lower Bound of Sorting

- For a list of n elements, there are $n!$ permutations. We suppose a and b are two elements of the list.
- A general comparison yields a decision out of two possibilities, say, it's $a < b$.
- The number of permutation containing a ahead of b is the same as the number of those containing b ahead of a. The decision eliminates the latter half of the permutations.
- To reach the sorted permutation in all cases, we need at least $\log(n!)$ steps. This is the lower bound for sorting using only the comparisons of two elements.

$a < b$	d,b,c,a	d,b,a,c	d,c,b,a	d,c,a,b	d,a,b,c	d,a,c,b
$b < c$	c,b,a,d	c,b,d,a	c,a,b,d	c,a,d,b	c,d,b,a	c,d,a,b
$c < d$	b,a,c,d	b,a,d,c	b,c,a,d	b,c,d,a	b,d,a,c	b,d,c,a
	a,b,c,d	a,b,d,c	a,c,b,d	a,c,d,b	a,d,b,c	a,d,c,b

A General Lower Bound of Sorting

- For a list of n elements, there are $n!$ permutations. We suppose a and b are two elements of the list.
- A general comparison yields a decision out of two possibilities, say, it's $a < b$.
- The number of permutation containing a ahead of b is the same as the number of those containing b ahead of a. The decision eliminates the latter half of the permutations.
- To reach the sorted permutation in all cases, we need at least $\log(n!)$ steps. This is the lower bound for sorting using only the comparisons of two elements.

~~$a<b$~~	~~d,b,c,a~~	~~d,b,a,c~~	~~d,c,b,a~~	d,c,a,b	d,a,b,c	d,a,c,b
$b<c$	~~c,b,a,d~~	~~c,b,d,a~~	c,a,b,d	c,a,d,b	~~c,d,b,a~~	c,d,a,b
$c<d$	~~b,a,c,d~~	~~b,a,d,c~~	~~b,c,a,d~~	~~b,c,d,a~~	~~b,d,a,c~~	~~b,d,c,a~~
	a,b,c,d	a,b,d,c	a,c,b,d	a,c,d,b	a,d,b,c	a,d,c,b

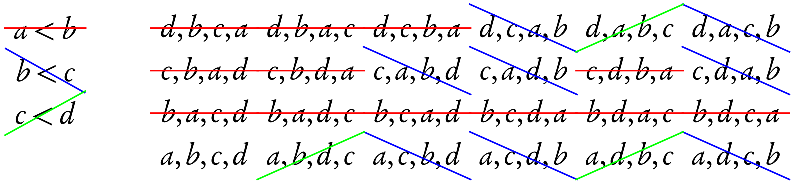
A General Lower Bound of Sorting

- For a list of n elements, there are $n!$ permutations. We suppose a and b are two elements of the list.
- A general comparison yields a decision out of two possibilities, say, it's $a < b$.
- The number of permutation containing a ahead of b is the same as the number of those containing b ahead of a. The decision eliminates the latter half of the permutations.
- To reach the sorted permutation in all cases, we need at least $\log(n!)$ steps. This is the lower bound for sorting using only the comparisons of two elements.

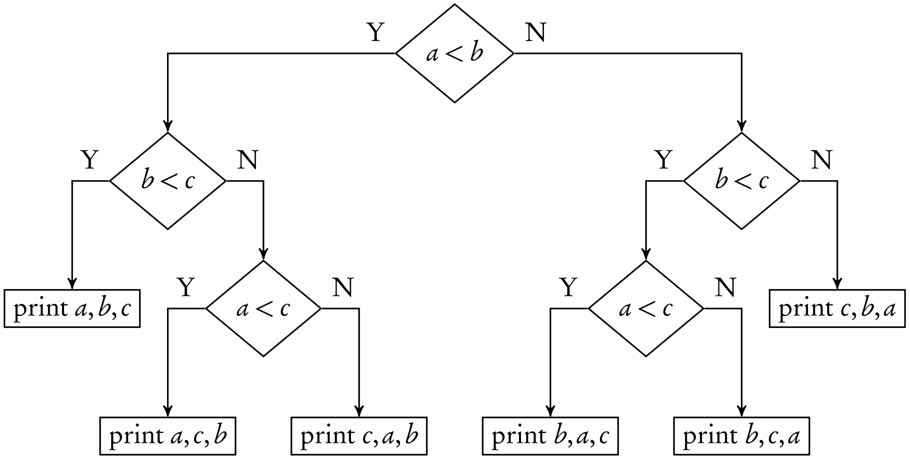
~~$a < b$~~	~~d,b,c,a~~	~~d,b,a,c~~	~~d,c,b,a~~	$\cancel{d,c,a,b}$	d,a,b,c	$\cancel{d,a,c,b}$
$\cancel{b < c}$	~~c,b,a,d~~	~~c,b,d,a~~	$\cancel{c,a,b,d}$	$\cancel{c,a,d,b}$	~~c,d,b,a~~	$\cancel{c,d,a,b}$
$c < d$	~~b,a,c,d~~	~~b,a,d,c~~	~~b,c,a,d~~	~~b,c,d,a~~	~~b,d,a,c~~	~~b,d,c,a~~
	a,b,c,d	a,b,d,c	$\cancel{a,c,b,d}$	$\cancel{a,c,d,b}$	a,d,b,c	$\cancel{a,d,c,b}$

A General Lower Bound of Sorting

- For a list of n elements, there are $n!$ permutations. We suppose a and b are two elements of the list.
- A general comparison yields a decision out of two possibilities, say, it's $a < b$.
- The number of permutation containing a ahead of b is the same as the number of those containing b ahead of a. The decision eliminates the latter half of the permutations.
- To reach the sorted permutation in all cases, we need at least $\log(n!)$ steps. This is the lower bound for sorting using only the comparisons of two elements.


$a < b$
$b < c$
$c < d$
d,b,c,a d,b,a,c d,c,b,a d,c,a,b d,a,b,c d,a,c,b
c,b,a,d c,b,d,a c,a,b,d c,a,d,b c,d,b,a c,d,a,b
b,a,c,d b,a,d,c b,c,a,d b,c,d,a b,d,a,c b,d,c,a
a,b,c,d a,b,d,c a,c,b,d a,c,d,b a,d,b,c a,d,c,b

A Decision Tree of Three Elements



Approximating $\log(n!)$

$$
\begin{aligned}
\log(n!) &= \log(n(n-1)(n-2)\cdots 2\cdot 1) \\
&= \log n + \log(n-1) + \cdots + \log 2 + \log 1 \\
&\geqslant \underbrace{\log n + \log(n-1) + \cdots + \log\frac{n}{2}}_{\frac{n}{2}\text{ items}} && \text{omit the latter half items} \\
&\geqslant \underbrace{\log\frac{n}{2} + \log\frac{n}{2} + \cdots + \log\frac{n}{2}}_{\frac{n}{2}\text{ items}} && \text{decrement each item to } \frac{n}{2} \\
&= \frac{n}{2}\log\frac{n}{2} = \frac{n}{2}(\log n - 1).
\end{aligned}
$$

Therefore, $\log(n!) = \Omega(n\log n)$ is the lower bound of the worst-case time complexity of any sorting algorithm based on comparisons.