Search and Sort Algorithms

Sven Pfiffner

November 19, 2018

Contents

1 Search **1**
- 1.1 Binary Search . . . 2
- 1.2 Interpolation Search . . . 4
- 1.3 Exponential Search . . . 5
- 1.4 Linear Search . . . 7

2 Sort **8**
- 2.1 Bubblesort . . . 8
- 2.2 Heapsort . . . 10
- 2.3 Mergesort . . . 13
- 2.4 Quicksort . . . 16

3 Cheatsheet **19**

For the sake of simplicity this summary assumes that an array starts with index 1 (Except for the Implementation part)

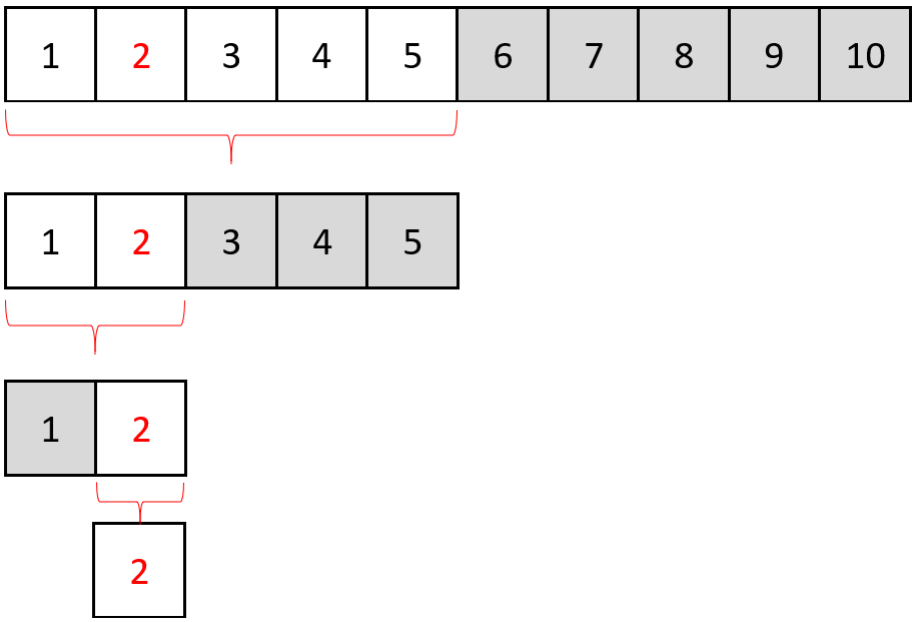
1 Search

If we speak of searching an item in an array, we mean that we want to know whether said item exists and if so, its location (i.e. its index in the array). In general there are two possible situations: Search in an ordered array and search in an unordered array. All of the search-algorithms covered in this summary, besides the linear search, only work if used on ordered arrays.

1.1 Binary Search

Theory The idea behind binary search is to divide the given array into smaller parts and search for the wanted key within them; which is essentially a divide and conquer approach.
Let A be an array with n elements in which we try to find the key k. We consider the key b at index $\lfloor n/2 \rfloor$.

- If $k = b$, then we found b and can return it.
- If $k < b$, then we must repeat the above on the subarray $A[1 : b]$.
- If $k > b$, then we must repeat the above on the subarray $A[b : n]$.



Binary Search for 2

Runtime Assume an array with n elements. For the amount of operations we get the following recurrence relation:

$$
\begin{aligned}
T(1) &= O(1) \\
T(n) &= T(\frac{n}{2}) + O(1) \\
&= T(\frac{n}{4}) + O(1) + O(1) \\
&= T(\frac{n}{8}) + O(1) + O(1) + O(1) \\
&= T(\frac{n}{2^i}) + i * O(1) \\
&= O(1) + log_2(n) * O(1) \\
&\in \underline{\underline{O(log(n))}}
\end{aligned}
$$

Implementation

```
int binarySearch(int[] A, int target) {
   int i = 0;
   int j = A.length-1;
   while(i <= j) {
      int center = (i+j)/2;
      if(A[center] == target) {return center;}
      else if (A[center] > target) {j = center--;}
      else {i = center++;}
   }
   return -1;
}
```

1.2 Interpolation Search

Theory Assume that we search for a key which is expected at a specific location in the array. In this case it does not make sense to divide the array into two almost equal sized sub-arrays as we do with the Binary-search approach.

Imagine an array with n elements in which we try to find the key k, which is close to the last element. In this case it is reasonable to start search towards the end side instead of the center as we would with binary-search.

Runtime This algorithm is only an improvement compared to binary search if used on an uniformly distributed array.

- Let A be uniformly distributed: $\underline{\underline{O(log(log(n)))}}$
- Let A have an arbitrary distribution: $\underline{\underline{O(n)}}$

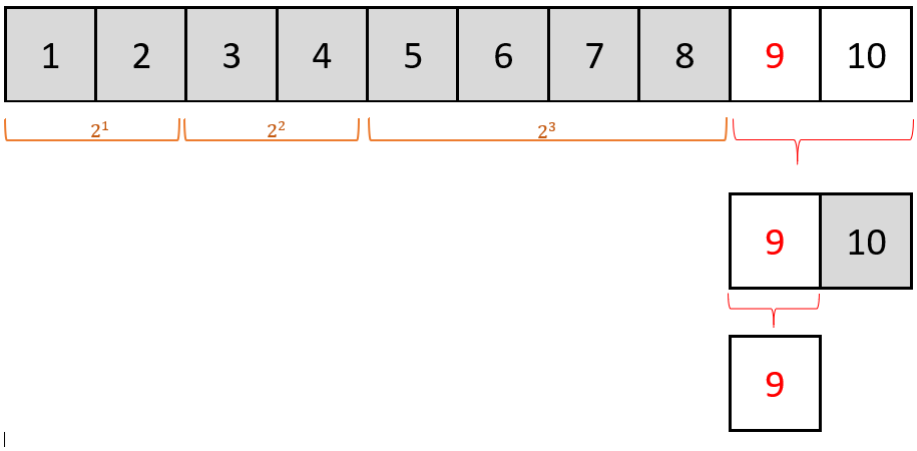
Implementation

```
int interpolationSearch(int[] A, int target) {
   int i = 0;
   int j = A.length-1;
   int cutPosition = -1;
   int delta = -1;
   while(i <= j && target >= A[i] && target <= A[j]) {
      delta = (target - A[i])/(A[j]-A[i]);
      cutPosition = i + (j-i) * delta;

      if(A[cutPosition] == target) {return cutPosition;}
      if(A[cutPosition] < target) {i = cutPosition ++;}
      else {j = cutPosition--;}
   }
   return -1;
}
```

1.3 Exponential Search

Theory Assume that we search for a key which is expected somewhere at the beginning of the array. The question that arises is whether there is a faster approach than binary-search, without having to deal with the worst-case-linear-runtime of interpolation-search.
Let b be the wanted key in the array A and $r = 1$ be the right border of the array search area. We can double r until:

- $A[r] = b$: In this case we found b at position r.
- $r > A.length$: In this case we can perform binary search on the subarray $A[\frac{r}{2} : A.length]$
- $r > A[r]$: in this case we can perform binary search on the subarray $A[\frac{r}{2} : r]$



Exponential Search for 9

Runtime It can easily be seen that the subarray which contains b can be found in $O(log(n))$ since we exponentialy increase r. As seen in [1.1] binary search takes $O(log(n))$ as well.

Assume $A.length = n$ and exponential search finds that b lies in the subarray $A[x, y]$. Since $A[x, y].length \leq n$, binary search on $A[x, y]$ takes $O(log(m)) \leq O(log(n))$. This leaves us with a total runtime of $O(log(n)) + O(log(m)) = \underline{\underline{O(log(n))}}$.

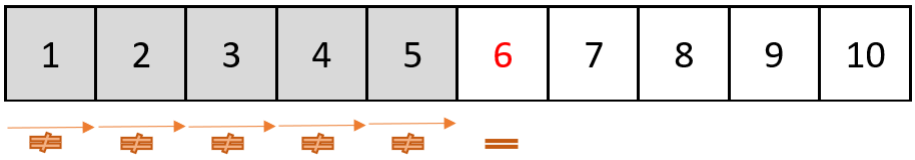
Since both binary search and exponential search have the same O-notation they have the same worst-case-runtime. However, since we assumed that b is somewhere at the beginning of the array, exponential search is most likely finding $A[x, y]$ and therefore b much faster than binary search.

Implementation

```
int exponentialSearch(int[] A, int target) {
   int r = 0;
   int index = 0;
   while(A[r] < target || r < A.length) {
      index++;
      r = 2**index;
   }
   binarySearch(A[r/2:min(r,A.length)],target)
}
```

1.4 Linear Search

Theory The most trivial approach to find a key b in an array A is to iterate through each element of A and compare with b. The advantage of this method is that it does not assume that the elements of the array are sorted and thus works with unordered arrays.



Linear Search for 6

Runtime Comparison between two elements takes $O(1)$. Let there be n elements in A. In the worst case b is the last element of A and thus n comparisons have to be made. This leaves us with:

$$n * (O(1)) = \underline{\underline{O(n)}}$$

Implementation

```
int linearSearch(int[] A, int target) {

   for(int i = 0; i<A.length; i++){
      if(A[i] == target) {
         return target;
      }
      return -1;
   }
}
```

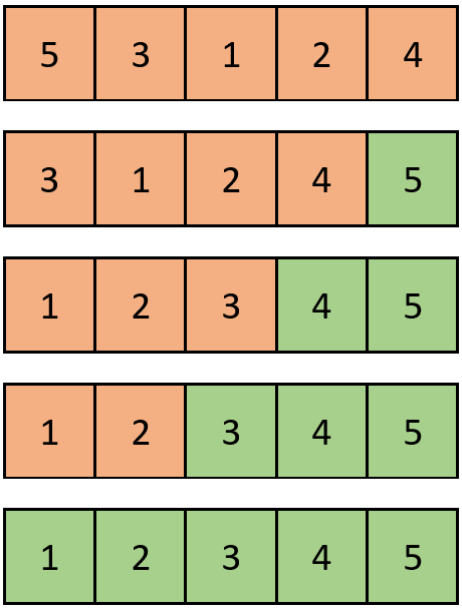
2 Sort

Processing of data is much easier and, in most cases, faster if it is sorted. (E.g. searching in an array, as seen in 1). However, sorting elements can be extremely expensive and complicated without a good strategy. The following Algorithms all sort an Array A with n elements such that bigger elements come before the lower ones; or more formally

$$\forall_i \forall_j (A[i] \leq A[j] \iff i < j)$$

2.1 Bubblesort

Theory If $A[i] > A[j]$ for any $(i, j) | i < j$ then A is not sorted. If we iterate through an array and swap $x = A[i]$ and $y = A[i + 1]$ whenever $x > y$ we end up with the biggest element at the end of the array. This can be repeated and the last two elements of A are sorted. As one can see A should be sorted after $n - 1$ iterations of the above.



Bubblesort process on example array

Runtime Swapping two keys in an array takes $O(1)$. For each of the $n-1$ iterations at most n keys must be swapped. As a consequence:

$$\begin{aligned}(n-1)*(n*O(1)) &= (n-1)*O(n)\\ &= \underline{\underline{O(n^2)}}\end{aligned}$$

Implementation

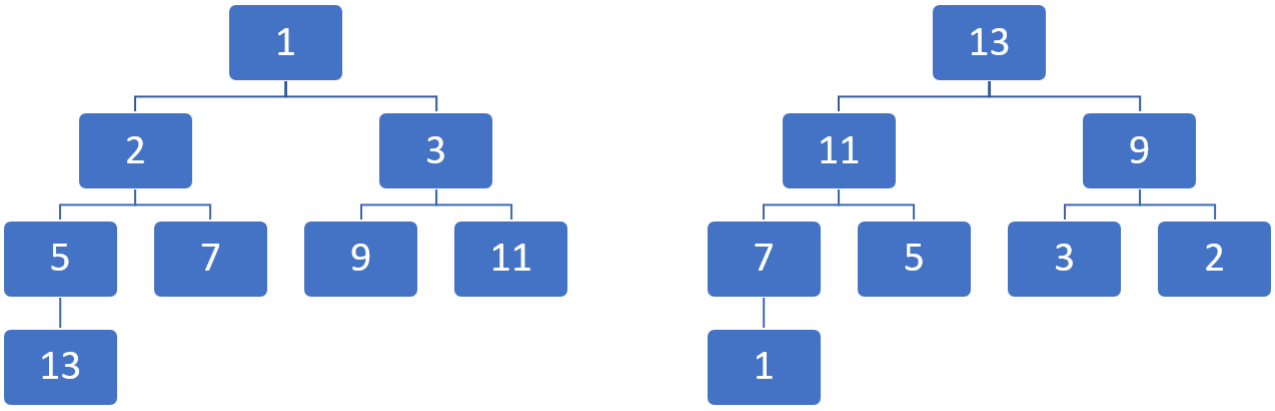
```
int[] bubbleSort(int[] A) {
   for(int i = 0; i<A.length; i++) {
      for(int j=0; j<A.length; j++) {
         if(A[i] > A[i+1]) {
            int container = A[i+1];
            A[i+1] = A[i];
            A[i] = container;
         }
      }
   }
   return A;
}
```

2.2 Heapsort

Theory In the bubblesort approach we swapped keys to bring the biggest key in an array to its end. However, we actually just need one swap of keys to do so (by finding the biggest key in A and swapping it with the last key in A). To do so, we have to find the biggest key in A which can be done in $O(n)$ and swap it in $O(1)$. By repeating this on the resulting unordered subarray until the array is sorted, we solve the problem in $O(n^2)$. This however, is no improvement to bubblesort, so the question arises whether we can improve the efficiency of the steps.

Assume an array whose keys are all in the wrong place. We have to swap n-times to order the array, no mather which algorithm we choose. As consequence we can not improve the amount of swaps since it will always be $O(n)$ at worst case.

What can be improved though, is the process of finding the biggest element in an array. This can be archived with a special data structure: a so called **MaxHeap** that provides the biggest key of A in $O(1)$.



Example of a MinHeap (left) and MaxHeap (right)

Heapsort is a non stable sorting algorithm

Runtime Assume we build a MaxHeap out of all keys in A. To do so we have to read every key into a BinaryTree ($O(n)$) and restore MaxHeap-condition on the BinaryHeap, which means that the sucessors of each key mustn't be greater than the key itself. Restoring the MaxHeap-condition can be done in:

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= T(\frac{n}{2}) + O(1) \\ &\in \underline{\underline{O(log(n))}} \end{aligned}$$

To sort the array we have to:

i Build a MaxHeap: $O(n)$

ii Repeat n times:

- Restore MaxHeap condition: $O(log(n))$
- Extract Max $O(1)$
- Remove Max $O(1)$

This leaves us with a total runtime of

$$\begin{aligned} O(n) + n * (O(log(n)) + O(1) + O(1)) &= 3 * O(n) + O(n) * O(log(n)) \\ &= O(n) * O(log(n)) \\ &= \underline{\underline{O(n * log(n))}} \end{aligned}$$

Implementation

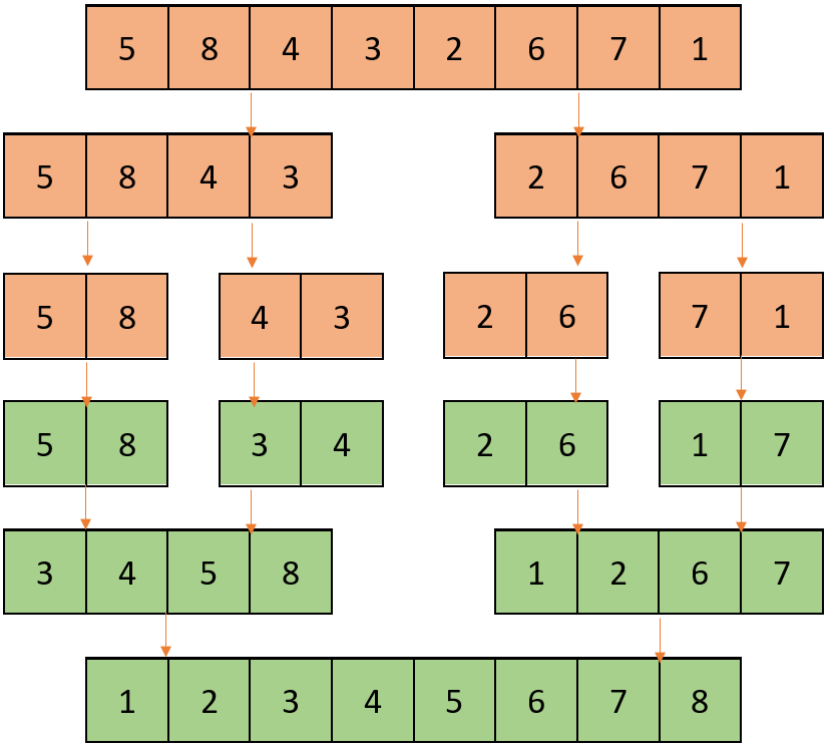
```
int[] heapSort(int[] A) {
   for(int i = A.length/2; i > 0; i--) {
      restoreHeapCondition(A, i, A.length);
   }

   for(int m = A.length; i > 1; m--) {
      int container = A[0];
      A[0] = A[m];
      A[m] = container;
      restoreHeapCondition(A, 0, m-1);
   }
   return A;
}
```

```
void restoreHeapCondition(int[] A, int i, int m) {
   while(2*i <= m) {
      int j = 2*i;
      if(j + 1 <= m && A[j] < A[j+1]) {
         j++;
      }
      if(A[i] >= A[j]) {
         return;
      }
      int container = A[i];
      A[i] = A[j];
      A[j] = A[i];
      i = j;
   }
}
```

2.3 Mergesort

Theory The idea behind mergesort is to sort an array by divide-and-conquer. We divide A at $\lfloor \frac{A.length}{2} \rfloor$ and sort the resulting subarrays recursively as follows:

- Let A_x and A_y be two sorted subarrays of A. We create a new Array B of length $A_x.length$ + $A_y.length$ and set a pointer (i, j) to the beginning of each subarray. Since A_x and A_y are already internally sorted, we only have to deside whether $A_x[i]$ or $A_y[j]$ is bigger. We add the bigger key to the next empty place of B and increase the corresponding pointer by one. If either A_x or A_y is fully traversed, we append the remaining keys of the other subarray to B. This way B becomes an ordered Array that contains all elements of A_x and A_y.



Mergesort process on example array

Runtime

- The divide step computes the midpoint by dividing the array length which takes $\Theta(1)$.
- The conquer step sorts two subarrays of $\approx 2/n$ elements each
- Merge merges n elements in $\Theta(n)$.

This gives us the following recurrence relation:

$$\begin{aligned} T(n) &= T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) + \Theta(1) \\ &= \underline{\underline{\Theta(n * log(n))}} \end{aligned}$$

Implementation

```
void mergeSort(int[] A, int left, int right) {
   if(left < right) {
      int middle = (left + right)/2;
      mergeSort(A, left, middle);
      mergeSort(A, middle + 1, right);
      merge(A, left, middle, right);
   }
}
```

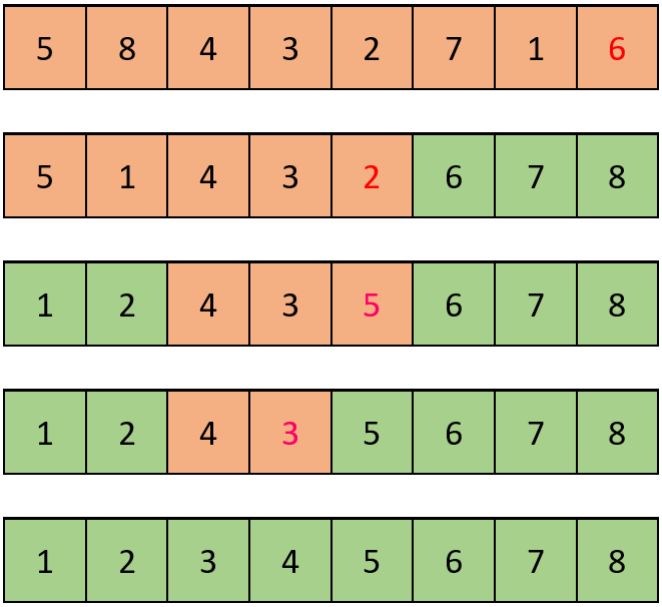
```
void merge(int[] A, int left,int middle, int right) {
   int[] B = new int[right-left+1];
   int i,j,k = left,middle+1,1;

   while(i <= middle && j <= right) {
      if(A[i] <= A[j]) {
         B[k] = A[i];
         i++;
      }
      else {
         B[k] = A[j];
         j++;
      }
      k++;
      while(i <= middle) {
         B[k] = A[i];
         i++;
         k++;
      }
      while(j <= right) {
         B[k] = A[j];
         j++;
         k++;
      }
      for(k = left; k==right; k++) {
         A[k] = B[k - left + 1];
      }
   }
}
```

2.4 Quicksort

Theory A disadvantage of mergesort is that it needs $O(n)$ of extra space for the merge step and therefore is not "in-place". The idea behind Quicksort is to divide A into two subarrays A_x, A_y such that $\forall_i|(A_x[i] \leq A_y[i])$ and thus neglect the merge step.

This can be achieved by dividing the array at an arbitrary key p and order it such that all elements smaller or equal to p are left of it and those greater than p are right of it; or more formally

$$\forall_i|((i \leq p.index) \iff (A[i] \leq p) \wedge ((i > p.index) \iff (A[i] > p))$$



Quicksort process on example array

Quicksort is a non stable sorting algorithm

Runtime Runtime of quicksort depends strongly on which p is chosen.

- Best case is that the amount of keys left to p is nearly the same as right to p. This way we get

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(\frac{n}{2}) + c * n \\ &= \underline{\underline{O(n * log(n))}} \end{aligned}$$

- Worst case is that we choose $A.min$ or $A.max$ as p and thus create a subarray of length 0 and one of length $n - 1$. This way we get

$$\begin{aligned} T(1) &= 0 \\ T(n) &= T(n-1) + c * n \\ &= \underline{\underline{\Theta(n^2)}} \end{aligned}$$

While quicksort is $O(n^2)$ at worst-case, experience shows that its runtime on average is actually $O(n * log(n))$

Implementation

```
void quickSort(int[] A, int left, int right) {
   if(left < right) {
   int k = partition(A,left,right);
   quickSort(A,l,k-1);
   quickSort(A,k+1,r);
   }
}
```

```
int partition(int[] A, int left, int right) {
   int i = l;
   int j = r-1;
   int p = A[r];
   int container;
   do {
      while(i<r && A[i] < p) {i++;}
      while(j > l && A[j] >p) {j--;}
      if (i < j) {
         container = A[i];
         A[i] = A[j];
         A[j] = A[i];
      }
   } while (i < j);
   container = A[i];
   A[i] = A[j];
   A[j] = A[i];
   return i;
}
```

3 Cheatsheet

Search Algorithms			
Name	Best-Case	Worst-Case	Average
Binary Search	$\Omega(1)$	$O(log(n))$	$\Theta(log(n))$
Interpolation Search	$\Omega(1)$	$O(n)$	$\Theta(log(log(n)))$
Exponential Search	$\Omega(1)$	$O(log(n))$	$\Theta(log(n))$
Linear Search	$\Omega(1)$	$O(n)$	$\Theta(\frac{n}{2})$

Sort Algorithms					
Name	Best-Case	Worst-Case	Average	In-place	Stable
Bubblesort	$\Omega(n)$	$O(n^2)$	$\Theta(n^2)$	✓	✓
Heapsort	$\Omega(n * log(n))$	$O(n * log(n))$	$\Theta(n * log(n))$	✓	×
Mergesort	$\Omega(n * log(n))$	$O(n * log(n))$	$\Theta(n * log(n))$	×	✓
Quicksort	$\Omega(n * log(n))$	$O(n^2)$	$\Theta(n * log(n))$	✓	×

References

[1] Robert Sedgewick, Kevin Wayne: *Algorithms Fourth Edition*. Addison Wesley, Boston, 2011.

[2] John Paul Mueller, Luca Massaron: *Algorithms For Dummies*. John Wiley and Sons, New Jersey, 2017.

[3] Michael Soltys: *An introduction to the analysis of algorithms* World Scientific, Singapore 2012.

[4] [DE] Thomas Ottmann, Peter Widmayer: *Algorithmen und Datenstrukuren*. Spektrum, Heidelberg, 5. Auflage, 2012.