Algortihms and datastructures

Kristoffer Klokker

2022

Contents

1 Algorithm analyse

An algorithm must stop for all input and give the correct output.
An algorithms quality is determined from:

- Speed
- Memory used
- Complexity of implmentation
- Other use cases like stability

1.1 Algorithm speed

The measure speed and memory the worst case of the algorithm is always used due to its simplicity in calculations and gurantee.
The measurement is used as a function of input size using the big O notation, which says for a given input with n size it will take n^2 time to run for example. In reality the speed of an algorithm is not equivalent to the input size due to small constant in every operation but these are so miniscule, they should not be considered. A function may be tuned to have lower constants but this is more a topic for theoretical algorithm optimization.
When comparing algorithm speed the math relations symbols are replaced as such:

- $=-\Theta$ Theta
- $\leq - O$ O
- $\geq - \Omega$ Omega
- $<- o$ little o
- $>- \omega$ little omega

When comparing the speed of two algorithms the following methods can be used:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \qquad f(n) = \Theta(g(n))$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad f(n) = o(g(n))$$

The generel speeds of alogirthms in order is:

$$1,\ \log_2 n,\ \sqrt{n},\ n,\ n\log_2 n,\ n\sqrt{n},\ n^2,\ n^3,\ n^{10},\ 2^n$$

1.2 Ram-model

To analyze an algorithm a model is used most often the ram model. The ram model is a very simple interpretation of a computer.
The model consist of a CPU, memory and basic operation (add, sub, mult, shift, move, jump)
The time of the algorithm is then measured in amount of basic operations done.
The memory is determined as the amount of memory cell used.

2 Sorting algorithms

Sorting algorithms are used to sort an array of items in an ascending order.

2.1 Insertion sort

One of the most simple sorting algorithms.
Works by going through the list from index 1 and moves moves every entry before the element 1 up until the element is to the right of an element smaller than the element.
This will therefore have a run time of $O(n^2)$ due to the scenario where the array is in decending order where it will moves $n-1, n-2, .., 1$ elements which is $\sum_{j=1}^{n} j = \frac{(n-1)n}{2} \leq \frac{n^2-n}{2} = n^2$

```
1: Insertion-Sort(A, n)
2: for j = 1 to n do
3:     key = A[j]
4:     i = j − 1
5:     while i ≥ 0 and A[i] > key do
6:         a[i + 1] = A[i]
7:         i = i − 1
8:     end while
9:     A[i + 1] = key
10: end for
```

2.2 Selectionsort

Selectionsort is done by taking all elements and then searching for the smallest element and inserting it into the list.
This will result in the same run time $O(n^2)$ due to the same reasoning as

insertionsort.

```
1: Selection-Sort(A, n)
2: Array output = newArray(n)
3: i = 0
4: while i < n do
5:     output[i] = A[i]
6:     A[i].remove()
7:     i++
8: end while
```

2.3 Merge sort

Merge sort is an sorting algorithm which is based on the simple merge of two sorted list. Here a new list is created and for every element of the two list, is the lowest value choosen from the first place of the list and putted into the new list. This is is done until both list are empty and when sorting the undefined element is simply equal to infinity when compared.

To make this into an sorting algorithm, every element in an unsorted list is made into to list of length 1. Then merge is performed on the lists in pairs of two, from which new sorted lists emerges. This is done until a single lists is left with all the elements which is now sorted.

The speed of this sorting algorithm will therefore be $n\lceil \log_2 n \rceil$ due to for every level of merge there will be performed n operations. The amount of levels of merges are equal to $\log_2 n$ due to first there will be n lists, then $n/2$ lists, and then $n/2^2$. This will continue until $n/2^k = 1$ which rearranged to find k will be $log_2 n = k$.
The ceiling of $\log_2 n$ is due to in scenarios where the list length is not a potens of two and a list has no pair in this case it will go on to the next level and thus creating a new level once the list has 1 more item than a potens of two.

2.4 Quick sort

A recursive sorting algorithm which runs mostly at $O(n \log n)$ time but worst case is $O(n^2)$.
The sorting algorithm is smart due to it not needing subarrays copies and working in the original array.
The main idea, is choosing an element in the array, then sorting everything

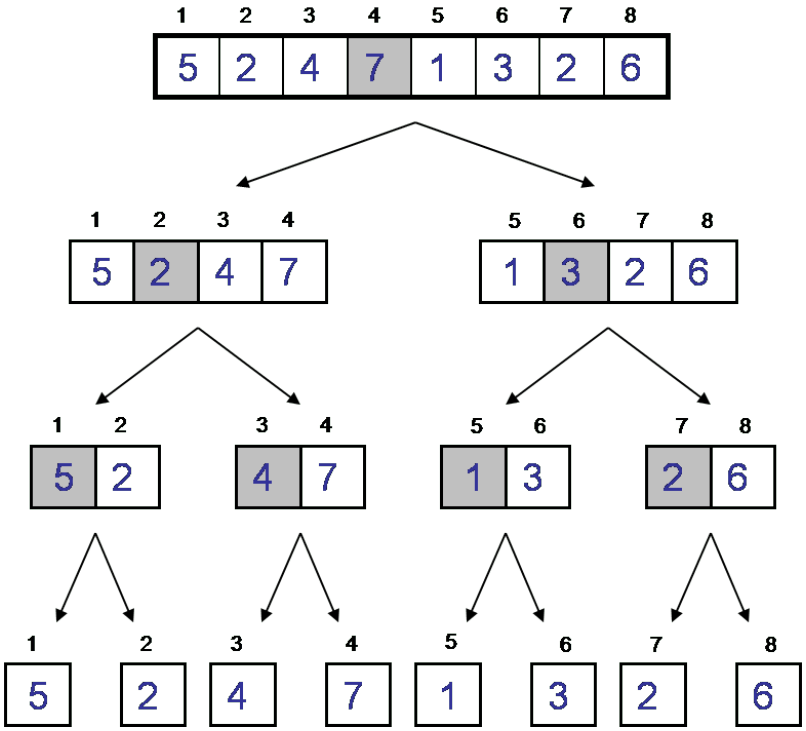

Figure 1: Merge sort list split up illustration from j2eereference.com

in two baskets lower or equal and higher.
Then the element is placed between the two baskets and quick sort is ran upon the two baskets.

```
1: Quick-Sort(A, p, r)
2: if r ≤ p then
3:     return;
4: end if
5: i = p
6: for j = p to r do
7:     if A[j] ≤ x then
8:         temp = A[i];
9:         A[i] = A[j];
10:        A[j] = temp;
11:        i = i + 1;
12:    end if
13: end for
14: temp = A[i];
15: A[i] = A[r];
16: A[r] = temp;
17: Quick − Sort(A, p, i − 1);
18: Quick − Sort(A, i + 1, r);
```

The arguments is the array A the start of the sort p and the end r.
Here it is seen that in the for loop if the current element is smaller than the choosen element it is switched with the first element in larger basket such that the smaller basket is in front.
In case the element is larger nothing is done. Then at the end the choosen element is switched with the first element of the larger bucket.

2.5 Heap sort

Heap sort is based on the Heap and runs in $O(n \log n)$ and uses no extra space. The heap which is a binary tree which follows heap order. The heap order dictates that a parent should always be greater or equal to its children. The tree should also have heap shape, which is all branches should be the saem height with a maximum difference of 1. In case of a larger branch it should be on the left side of the tree.
This heap can is illustrated as a tree but is actually an array. Here the arrays first entry is the root, the roots children will then be at index 2 times index and 2 times index + 1
This means the parent is at $\lfloor i/2 \rfloor$ relative to the child. Heap sort then uses

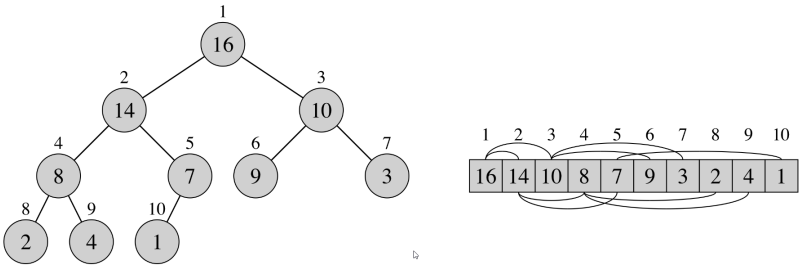

Figure 2: The heap illustration and the heap array

the heap array, and when sorting the first entry or the root is switched with the last entry of the heap array.
By this the size of the heap array shrinks with 1 and the rest of the array is the sorted array.
When switched the heap is reconstructed.
The reconstruction process is as follows:

1: Max-Heapify(A, i, n)
2: $l = 2i$;
3: $r = 2i + 1$;
4: **if** $l \leq n$ and $A[l] > A[i]$ **then**
5: $\quad largest = l$;
6: **else**
7: $\quad largest = i$;
8: **end if**
9: **if** $r \leq n$ and $A[r] > A[largest]$ **then**
10: $\quad largest = r$;
11: **end if**
12: **if** $largest! = i$ **then**
13: $\quad temp = A[i]$;
14: $\quad A[i] = A[largest]$;
15: $\quad A[largest] = temp$;
16: $\quad Max - Heapify(A, largest, n)$;
17: **end if**

The n is used as the length of the heap and i is the current node to check. The first if check if the left child is larger, the second checks if the right child is larger, and the third check if something should be changed and if then call itself upon the newly changed child.
With this function in mind heap sort becomes:

1: HeapSort(A, n)
2: Max-Heapify(A,n);

3: **for** $i = n$; $2 < i$; $i - -$; **do**
4: $temp = A[i]$;
5: $A[i] = A[0]$;
6: $A[0] = temp$;
7: Max-Heapify$(A, 1, i - 1)$;
8: **end for**

The run time of this is $O(n \log n)$ dut to the trees height will maximally be $\log n$ which will be the number of exhanges n times.

2.6 Counter sort

Counter sort works in $O(n + k)$ time on arrays of objects represented by integers by having an array with k length. Where k is the highest integer.
It then goes through the orginal array and takes every elements value and uses it as index for the k array and itterate by 1.
Then a new array with length n is made and every element in the k array is added to the new array with the value of elements index as many times as the value of element.

2.7 Radix sort

Radis sort runs in $O(d(n + k))$ time, where d is amount of digits in largest number and k is the max value of the base of the number system. First Counter sort is done upon every elements lasts digit. Then from the new array counter sort is done again on the second lasts digit. This is repeated as many times as the most digits in the largest number.
This therefore means in the counter sort, the k array only has the length the max value of the number system and instead of itterate it is now a list with references.
This therefore solves the issue of now knowing the largest digit, and the temperary array is a lot shorter.

3 algorithms

3.1 Divide-and-Conquer algorithms

The same as recursive algorithms.
When illustrated a tree is often used to describe the calls and calls backs

3.2 Lower bound of comparison sorting algorithms

To find the lower bound of comparison sorting algorithms, the comparison model can be used.
The model consist of a binary tree where every branch is a comparison, and the leafs are possible answers.
The run time here is the comparisons, therefore the run time to a leaf is the length of the path. Therefore making worst case height of tree.
It can therefore be seen when sorting an array of n elements there are $n!$ different rearrangement.
Each rearrangement needs a leaf therefore the tree has to have $n!$ leafs.
The maximum amount of leafs in a tree is 2^h where h is the height.
Therefore the minimum amount of height will be $2^h \geq n! \rightarrow h \geq \log n!$

$$
\begin{aligned}
h &\geq \log n! \\
&= \log(n \cdot (n-1) \cdot (n-2) \cdot ...) \\
&= \log n + \log(n-1) + ... \\
&\geq \frac{n}{2} \cdot \log(\frac{n}{2}) \\
&= \frac{n}{2}(\log(n) - 1)
\end{aligned}
$$

It can here be derived via the summation, that $h = \Omega(n \log n)$

3.3 Maximum summation of sequential elements in array

The most simple version of this, is two pointers with start of seqence and end, where for each element in the array the start is appointed from which all possible ends if summed and the largest value is taken.
This means the algorithm will take $O(n^3)$ due to first every element being appointed as start, from which all elements is tested as the end of sequence, from which the summation is done. This will round up to the worst case being $O(n^3)$
This can be optimized to a run time of $O(n)$, by instead of recalculating everything for every pointer a pointer is started at the start from which an end pointer is also at the start.
Here the largest summation is found, if it is larger than the best sequence it is assigned. Then the end pointer is moved forward and the new element is summed to the recent best sequence. If the sequence summation is now

below zero the start pointer is moved to the end otherwise the end pointer is moved forward.

```
1: MaxSequence(A)
2: max = 0
3: tempMax = 0
4: i = 0
5: while i < A.length do
6:     tempMax = max(tempMax + A[i], 0)
7:     max = max(max, tempMax)
8:     i++
9: end while
```

4 Datastructures

4.1 Priority queues

Priority queues are based on the heap structure, which is described in Heap sort.
Here the queue objects are given a key from which the value iindicates the priority of the object.
It can here selected if the wanted heap is a max heap or min heap, but most often a max heap.
For the queue to work it uses the following functions:

- Max-Heapify(A,i,n) - makes the array in max heap order with the length n and looks at node i
- Heap-Extract-Max(A,n) - returns the highest priority object and remove from heap
- Heap-Increase-Key(A,i,key) - modifies the key i to a higher value key and place it correctly in heap
- Max-Heap-Insert(A,key) - Inserts key into the heap

4.1.1 Pseudo code

Max-Heapify code

```
1: Heap-Extract-Max(A, n)
2: if n < 1 then
3:     error 'Heap underflow'
4: end if
```

5: $max = A[0]$
6: $A[0] = A[n-1]$
7: $Max - Heapify(A, 1, n-1)$
8: return max

.

1: Heap-Increase-Key(A, i, key)
2: **if** $key < key[i]$ **then**
3: error 'new key is smaller than current'
4: **end if**
5: **while** $i > 0$ AND $A[Parent(i)] < A[i]$ **do**
6: $temp = A[i]$;
7: $A[i] = A[i/2]]$;
8: $A[i/2] = temp$;
9: $i = i/2$
10: **end while**

.

1: Max-Heap-Insert(A,key,n)
2: $A.insert(n, -\infty)$
3: $Heap - Increase - Key(A, n, key)$