Sorting Part 2

1 Sorting

1.1 Theorem

For any **comparison based** sorting algorithm $\mathcal{A}$ and any $n \in \mathbb{N}$ large enough there exists an input of length n that requires $\mathcal{A}$ to perform $\Omega(n \log n)$ comparisons.

This is an example of a general lower bound: no such algorithm, past, preset or future can consistently beat the stated bound

Two observations

1. The theorem talks about comparison based sorting algorithms. The vast majority of sorting algorithms are in that class
2. Theorem says that "there exists an input". That means that for most inputs of length n it may actually beat the bound, but that there must be at leat one for which it does not

1.2 Bucketsort

```
BucketSort($a_1, ..., a_n \in \{0, ... K-1\}, n \geqslant 2$)
create array C[0,...,K-1] and initialise each entry C[i] to zero
for i=1 to n do
    increment value C[$a_i$] by one
end for
for i=0 to K-1 do
    for j-1 to C[i] do
        print i
    end for
end for
```

This has running time $O(n + K)$.
This means that if K is small, say $o(n \log n)$ the overall running time of bucket sort is $o(n \log n)$
The reason this beats the running time is that bucket sort doesn't do any comparisons.

1.3 Radix sort

This has an obvious drawback that if the range of items is large then you need a large number of buckets.

The improvement is to not look at the item values but "one level below"
The idea of radix sort:

- have as many buckets as you've got different digits, that is, for base 10 you've got 10 of them
- Repeatedly bucket sort by given digit
- number of rounds will depend on values (the longer the base 10 representations, the more rounds), but number of buckets only depends on number of different digits

This has a running time of $\Theta(d \cdot n)$ where d is the maximum number of digits per item (rounds)

Comparing this to bucket sort

- RadixSort: $\Theta(n \cdot \log K)$
- BucketSort:$\Theta(n + K)$

This means that for different inputs different types are better:

- if K=n then $\Theta(n \log n)$ vs $\Theta(n) \rightarrow$ BucketSort Wins
- if $K = n^2$ then $\Theta(n \log n)$ vs $\Theta(n^2) \rightarrow$ RadixSort Wins
- if $K = 2^n$ then $\Theta(n^2)$ vs $\Theta(2^n) \rightarrow$ RadixSort Wins

2 Searching and Selecting

Suppose you're given n numbers in array $A[1, ...n]$
Further suppose numbers are in sorted order
To make things slightly simpler, assume the n numbers are pairwise distinct, that is, no two are the same
Finally, assume you're also given a number x that's equal to one of the n numbers above, that is

$$\exists index i \in 1, ..., n such that x = A[i]$$

For instance

$$n = 8 \quad A[1...8] = (2, 3, 5, 7, 11, 13, 17, 19) \quad x = 17$$

Goal: Devise an algorithm that finds position p of x in A

2.1 Trivial search

```
int trivial_search (int A[1...n], int x)
{
    p=1
    while (A[p] != x) do
        p=p+1
    end while
    return p
}
```

Notice that this needs the fact that x is equal to one of the A[i]

2.2 Binary Search

Same assumptions as before:

- Peek right into the middle of the given array, at position $p = \lceil n/2 \rceil$
- If A[p]=x then we're lucky and done, and can return p
- Otherwise, if X is greater than A[p], we may focus our search on the stuff to the right of A[p] and may completely ignore anything to it's left, because x is already bigger than A[p] and hence must be even bigger than A[1...p-1].
- Likewise, if x is smaller than A[p], we may focus our search on stuff to the left of A[p] and may completely ignore anything to its right.

  That's because x is already smaller than A[p] and hence must be even smaller than $A[p + 1...n]$.
  Hence, if the "top level" call was search(A,1,n,x) for "search within array A, between the indices of 1 and n for element x". We could now recursively call from within this function search, using the updated indices, like so:
  `search(A,1,p-1,x)`

```
int search (int A[1..n], int left, int right, int x)
{
    if (right == left and A[left] != x)
        handle error; leave function
    p = middle-index between left and right
    if (A[p] == x) then
    return p
    // here come the recursive calls (if x not yet found)
    if (x > A[p]) then
        return search(A,p+1,right,x) // in right half
    else // x < A[p]
        return search(A,left,p-1,x) // in left half
}
```

Initial call would be "`search(A,1,n,x)`"

This is called binary search because in each (unsuccessful) step it at least halves the search space (we end up with the remaining, interesting part of the array in which x is hiding).

So far, had assumed that:

- The input is sorted, and
- We're looking for the position p of an element of given value v

Now we're changing the set-up

- input is unsorted, and
- We're looking for value of the ith smallest element in the input (clearly, if input were sorted, it'd be trivial: return ith from the left)

2.3 Quickselect

```
if (left == right) then
    return A[left]
else
    // rearrange/partition in place
    // return value "pivot" is index of pivot element
    // in A[] after partitioning
    pivot = Partition (A, left, right)
    // Now:
    // everything in A[left...pivot-1] is smaller than pivot
    // everything in A[pivot+1...right] is bigger than pivot
    // the pivot is in correct position w.r.t. sortedness
    if (i == pivot) then
        return A[i]
    else if (i < pivot) then
        return QuickSelect (A, left, pivot-1, i)
    else // i > pivot
        return QuickSelect (A, pivot+1, right, i)
    end if
end if
```

This has the flaw that if the input is in sorted order and you always pick the right most element as pivot, then just as slow as QuickSort is when it's slow.

- The part (LOW/HIGH) that's being recursed into is just one element smaller than the current input is
- $T(n) = T(n-1) + O(n)$ which means $T(n) = \Theta(n^2)$

So actually can be rather a lot slower than sorting with e.g. MergeSort

This can be sped up to $O(n)$ by first randomising the input

2.4 Median of Medians

2.4.1 The algorithm

1. If length(A)$\leqslant$5, then sort and return the ith smallest
2. Divide n elements into $\lfloor n/5 \rfloor$ groups of 5 elements each, plus at most one group containing the remaining n mod 5 elements
3. Find the median of each of the $\lceil n/5 \rceil$ groups by sorting each one, and then picking median from sorted group elements
4. Call **select** recursively on set of $\lceil n/5 \rceil$ medians found above, giving median-of-medians, x
5. Partition the entire input around x. Let k be the number of elements on low side plus one
    - x is the kth smallest element, and
    - there are n-k elements on high side of the partition
6. if i=k, return x. Otherwise use Select recursively to find ith smallest element of low side if $i < k$, or (i-k)th smallest on high side if $i > k$