.

INFODS HW 1

Idan Grady en Simon Oliehoek

May 2021

1 Pannenkoeken

```
A = new Array;
sign = Sign(A[1] - A[0]);
for ( i = 2; i < A.Length; i++)
{
    // Skip any next number that is already in a sorted order
    new_sign = Sign(A[i] - A[i - 1]);
    if (new_sign != 0 && new_sign != sign)
    {
        // A incorrect order has been found.
        if (sign == 1 && (A[i] <= A[0]  A[i] >= A[i-1])  sign == -1
 && (A[i] >= A[0]  A[i] <= A[i - 1]))
        {
            // If number is a new highest or lowest, only one flip
is required.
            // In the worst case, the first two numbers A[0] and A
[1] are the lowest and highest in the A.
            A = Flip(A, i);
            total_flips++;
            sign = new_sign;
        } else {
            // Binary search
            left = 0;
            right = A.Length - 1;
            reversed = (A[0] - A[A.Length - 1] > 0);
            while (right > left + 1)
            {
                // Bit shift is the same as deviding by 2. Saves a
little time
                int mid = (left + right) >> 1;

                // Check if value is left or right.
                if (A[i] < arr[mid])
                {
                    if (reversed) { left = mid; } else { right =
mid; }
                } else {
                    if (reversed) { right = mid; } else { left =
mid; }
                }
            }
```

```
35              // Get position of the number closest to A[i] between 0
     and i.
36              pos = left;
37
38              // Place item 'i' at pos.
39              A = Flip(A, i + 1);
40              A = Flip(A, i - pos + 1);
41              A = Flip(A, i - pos);
42
43              sign = new_sign;
44              total_flips += 3;
45          }
46      }
47  }
```

Listing 1: Insert Sort variant

In the best case, the array is already sorted and the comparisons would be $O(n)$ to check at line 7 if any number is not ascending or descending as the previous two numbers. In this case the Flips would be 0.

In the worst case, the array is unsorted and the first two values are the min and max bound. Meaning the comparison on line 10 will always be false. In this case, the total flips would be $3 * n$, since Flip() is called three times.

If we assume that a $Flip()$ takes $O(1)$ time, then we can deduce that the time complexity from this algorithm would be O(n log(n)).$NormallyInsertionSortwouldbe$O($n^2$) because of the swaps, but in this case no iterative swaps are executed, only 3 $Flip()$'s which are $O(1)$.

Lets assume that each operation or comparison takes Ci time. Each row of the algorithm above has a number, counting down in descending order, so line 1 would be C1, the for-loop at line 3 would be C3a, C3b and C3c and at line 6 would be C6.

The time it takes to run this algorithm in the worst case would be:

$$\begin{aligned} C1 + C2 + C3a + (n+1) * C3b + \\ n(C3c + C6 + C7 + C10 + C19 + C20 + C21 + \\ 2\log(n) * (C22 + C25 + C28 + 0.5 * C30 + 0.5 * C32) + \\ C36 + C39 + C40 + C41 + C43 + C44) \end{aligned}$$

Total comparisons will be $n * \log_2(n)$

This is equal to $O(n * \log(n))$ Worst total flips would be $n * C39 + n * C40 + n * C41$ which would be $3 * n$

```
1  for (int i = 0; i < input.Length; i++)
2  {
3      // Array 0 to i has already been sorted
4      int lowest = i;
```

```
5       int lowest_val = input[i];
6       for(int j = i; j<input.Length; j++)
7       {
8           if (input[j] < lowest_val)
9           {
10              lowest = j;
11              lowest_val = input[j];
12          }
13      }
14
15      input = flipIntArrayAt(input, lowest);
16      input = flipIntArrayAt(input, lowest + 1);
17
18      total_flips += 2;
19  }
```

Listing 2: Selection Sort variant

Vanwege de for-loop op lijn 1 zal het aantal flips hier altijd $2 * n$ zijn. Maar er zit een doubele for-loop in, dus als we aannemen dat $Flip()$ een tijd van $O(1)$ heeft, dan zal het algoritme $O(n^2)$ snel zijn.

```
1   pannenkoekenQS(A, low, high)
2   {
3       total_flips = 0;
4       if (low < high)
5       {
6           pivot_pos, added_flips;
7           (A, pivot_pos, added_flips) = pannenkoekenQSPart(A, low,
        high);
8           total_flips += added_flips;
9
10          (A, added_flips) = pannenkoekenQS(A, low, pivot_pos - 1);
11          total_flips += added_flips;
12          (A, added_flips) = pannenkoekenQS(A, pivot_pos + 1, high);
13          total_flips += added_flips;
14      }
15
16      return (A, total_flips);
17  }
18
19  pannenkoekenQSPart(A,  low,  high)
20  {
21       pivot = A[low];
22       left_wall = low;
23       total_flips = 0;
24       add_flips;
25
26      for ( i = low + 1; i <= high; i++)
27      {
28          if (A[i] < pivot) {
29              (A, add_flips) = pannenkoekenSwap(A, i, left_wall + 1);
30              left_wall++;
```

```
31              total_flips += add_flips;
32          }
33      }
34
35      (A, add_flips) = pannenkoekenSwap(A, low, left_wall);
36      total_flips += add_flips;
37
38      return (A, left_wall, total_flips);
39  }
40
41  pannenkoekenSwap(A,  pos1,  pos2)
42  {
43      if (pos1 == pos2)
44      {
45          return (A, 0);
46      }
47      else
48      {
49          // Get left and right position
50          left = (pos1 < pos2) ? pos1 : pos2;
51          right = (pos1 < pos2) ? pos2 : pos1;
52
53          // Swap two positions
54          A = Flip(A, left + 1);
55          A = Flip(A, right + 1);
56          A = Flip(A, right);
57          A = Flip(A, right - 1);
58          A = Flip(A, right);
59          A = Flip(A, left + 1);
60
61          return (A, 6);
62      }
63  }
```

Listing 3: Quick Sort Variant

De hoeveelheid keer de functie op lijn 19 wordt uitgeroepen is ongeveer everedig met:

$$2 * \log_{\frac{4}{3}}(n)$$

Want als we vanuit gaan dat de split gebeurt bij een kwart van elk eind van de array een slechte split is, zal 50% van de tijd een goede split zijn waarbij na de split er een deel over blijft met minder of gelijk aan 3/4 van het gesplitte array.

A_n is een array n keer gesplit. Een snelle split: $A_i <= 3/4 * (A_{i-1})$ Dus voor Q snelle splits geldt voor de array grote $<= n * (3/4)^Q$

Het splitten stopt als $n * (3/4)^Q <= 1$

$$\begin{aligned} n * (3/4)^Q &<= 1 \\ \Rightarrow \log_{\frac{3}{4}}(1/n) &>= Q \\ \Rightarrow \log_{\frac{4}{3}}(n) &>= Q \end{aligned}$$

Maar elke split heeft maar een 50% kans om goed te zijn, dus de hoeveelheid splits nodig is:

$$\log_{\frac{4}{3}}(n) >= 0.5/Q$$
$$\Rightarrow 2 * \log_{\frac{4}{3}}(n) >= Q$$

Dus de vergelijkingen per key is $2 * \log_{\frac{4}{3}}(n)$. Dat geeft ons: $2n * \log_{\frac{4}{3}}(n)$ wat correspondeerd met $O(n * \log(n))$

De formule $2n * \log_{\frac{4}{3}}(n)$ voor waardes van $n > 1$ is altijd kleiner dan $n * \log_2(n)$ van Insert Sort, dus als $Flip()$ een $O(1)$ is, dan is Quick Sort beter.

2 Kvick Sört

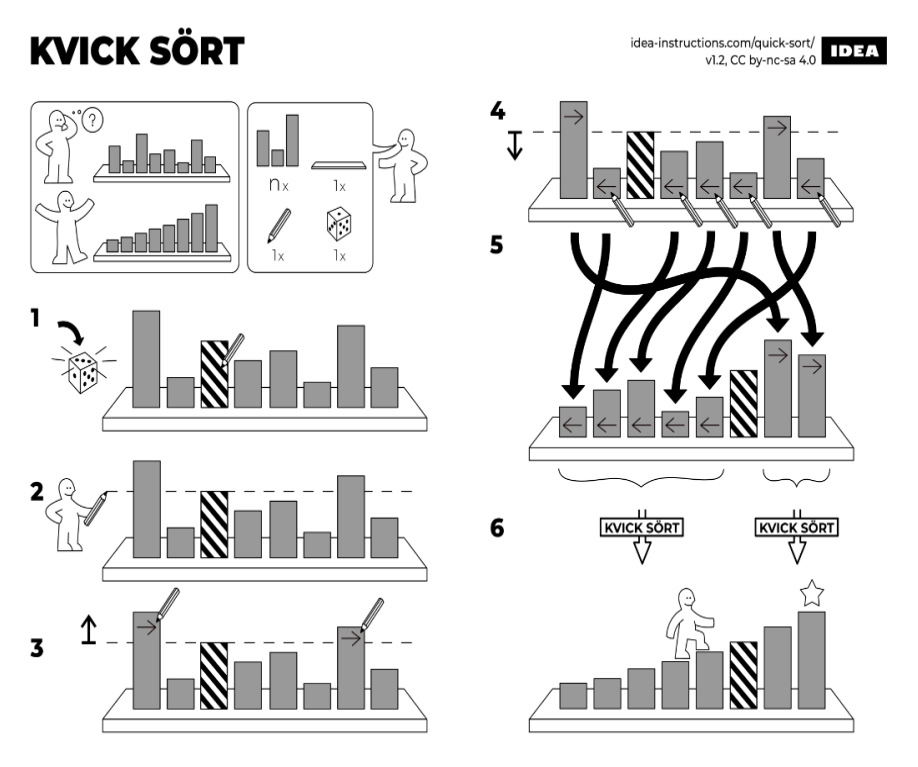

Figure 1: Kvick Sört

(a) Een gerandomiseerde pivotkeuze. Op stap 1 van figuur 1 wordt er een doppelsteen laten zien. Dit is bedoeld voor het bepalen van een willekeurig nummer.

(b) Elke balk wordt vergeleken met de pivot balk, behalve de pivot balk zelf.

$$f(n) = n - 1$$

(c) Als dit niet gebeurt, kan het zijn dat blokken met dezelfde lengte omwisselen. Dit kan uiteindelijk fouten geven. Bijvoorbeeld je hebt een array van objecten: $[2, a, 4, a, 4, b, 3, a, 1, a]$ en als je deze wilt sorteren kan het zo zijn dat je $4, a$ en $4, b$ niet omgewisselt wilt hebben. Als de blokjes op stap 5 van figuur 1 onderling omwisselen kan het algoritme dus niet zeker van zijn dat de volgorde van zelfde grotte blokken behouden blijft.

(d) Nee, in dit geval moet je kennis hebben over wat een dobbelsteen is, en dat de cijfers 1 tot 6 een volgorde van stappen bepalen, en dat de lengte van elk blokje een verschillende grootte bepaald (wat nog wel redelijk universeel door mensen begrepen kan worden). Je zal ook moeten snappen wat de pijlen betekenen. Ook is het van belang dat je het concept van n_x blokjes moet begrijpen, zodat je weet dat het werkt voor elk hoeveelheid van blokjes, en niet specefiek 8 blokjes van deze grootte.

3 Optellen

4 De Grote Omega

(a) Big Omega $\Omega(f(n))$ defines a running time of at least $e * f(n)$ for some constant e. This is to define the asymptotic lower bound.

$$g(n) = \Omega(f(n)) : \exists n_0, e : \forall n \geq n_0 \geq 0 : g(n) \geq e * f(n)$$

(b)

5 Een zware klus