

# Inhoudsopgave

<b>1</b>	<b>Inleiding</b>	<b>3</b>
1.1	What is an algorithm? . . . . .	3
1.2	Why study algorithms? . . . . .	4
1.3	Explain the convex hull algorithm . . . . .	4
1.4	Why analyze algorithms? . . . . .	4
1.5	Explain: Framework for analysing algorithms? . . . . .	4
1.6	What simplifications do we use when analyzing algorithms? . . .	5
1.7	What are the differences between BigO and $\sim$ notation? . . . . .	5
1.8	Examenvragen . . . . .	5
1.8.1	complexity charts . . . . .	5
1.8.2	$\sim$ notation . . . . .	6
<b>2</b>	<b>Sorting</b>	<b>6</b>
2.1	Explain Selection Sort . . . . .	6
2.2	Explain Insertion Sort . . . . .	7
2.3	Explain Merge Sort . . . . .	7
2.4	Compare the two sorting algorithm types . . . . .	7
2.5	Explain: Benadering van Stirling . . . . .	7
2.6	Examenvragen . . . . .	8
2.7	Oefeningen . . . . .	8
2.7.1	Tabel lezen . . . . .	8
2.7.2	Grafiek schetsen . . . . .	9
<b>3</b>	<b>QuickSort</b>	<b>9</b>
3.1	Explain QuickSort . . . . .	9
3.2	Give the first Partition variant . . . . .	10
3.3	Give the second Partition variant . . . . .	10
3.4	Give the third Partition variant: Lomuto . . . . .	11
3.5	Give some optimizations . . . . .	11
3.6	Examenvragen . . . . .	12
3.7	Oefeningen . . . . .	12
3.7.1	Bewijs dat de tijdscomplexiteit van Quicksort $1.39n \log n$ is. . . . .	12
3.7.2	Quicksort aanpassing . . . . .	13
3.7.3	Quicksort worst case . . . . .	14
<b>4</b>	<b>Sorteren in lineaire tijd</b>	<b>14</b>
4.1	Explain Bucket Sort . . . . .	14
4.2	How many buckets are needed and how are they sorted? . . . . .	15
4.3	Explain Key-Indexed (Counting) Sort . . . . .	15
4.4	What is an in-place sorting algorithm? . . . . .	15
4.5	What is a stable sorting algorithm? . . . . .	15
4.6	Explain LSD (radix) sort . . . . .	16
4.7	Explain MSD sort . . . . .	16

4.8	Explain 3-way string QuickSort . . . . .	16
4.9	What are some key lessons from sorting? . . . . .	17
4.10	Kan een niet-stabiel sorteeralgoritme stabiel gemaakt worden? . . . . .	17
4.11	Examenvragen . . . . .	17
4.12	Oefeningen . . . . .	19
4.12.1	Counting sort aanpassing . . . . .	19
<b>5</b>	<b>Stacks,Queues &amp; Hash Tabellen</b>	<b>19</b>
5.1	Explain a stack . . . . .	19
5.2	What are the differences between a stack implemented with linked list vs array? . . . . .	20
5.3	Explain a queue . . . . .	20
5.4	Explain hash tables . . . . .	20
5.5	What is Seperate Chaining? . . . . .	20
5.6	What is Linear Probing? . . . . .	21
5.7	Examenvragen . . . . .	21
5.8	Oefeningen . . . . .	22
5.8.1	Hashtabel . . . . .	22
<b>6</b>	<b>Priority Queues &amp; Balanced Trees</b>	<b>22</b>
6.1	Explain Priority Queues . . . . .	22
6.2	What is a Binary Tree? . . . . .	22
6.3	Explain Binary Heap (max/min heap) . . . . .	22
6.4	Explain Heapsort . . . . .	23
6.5	Explain Binary Search Trees (BST) . . . . .	23
6.6	Explain 2-3 Trees . . . . .	24
6.7	Explain Red-Black Trees . . . . .	25
6.8	Examenvragen . . . . .	25
6.9	Oefeningen . . . . .	27
6.9.1	Ternaire heap voor HeapSort . . . . .	27
6.9.2	Heap zonder boomstructuur . . . . .	27
6.9.3	Binaire Heap . . . . .	27
<b>7</b>	<b>Greedy Algoritmen</b>	<b>27</b>
7.1	What is a greedy algorithm? . . . . .	27
7.2	Explain run-length encoding . . . . .	27
7.3	Explain Huffman coding . . . . .	27
7.4	Prove Huffmans optimality . . . . .	28
7.5	Examenvragen . . . . .	29
7.6	Oefeningen . . . . .	30
7.6.1	Stel Huffman coderingsboom op . . . . .	30
7.6.2	Huffman coderingsboom voor Fibonacci getallen . . . . .	30
7.6.3	Aantal bits gecodeerde tekst . . . . .	31
7.6.4	Onder- en bovengrens afleiden . . . . .	31

<b>8</b>	<b>Minimum spanning tree and shortest path</b>	<b>32</b>
8.1	Explain Minimum Spanning Tree (MST)	32
8.2	Explain the cut property	32
8.3	Explain greedy MST algorithm	32
8.4	Explain Prim's Algorithm	32
8.5	Explain Prim's Lazy implementation	32
8.6	Can you improve Prim's implementation?	33
8.7	Explain Kruskal's algorithm	33
8.8	Explain Shortest Path (SPT)	34
8.9	Explain Dijkstras algorithm	34
8.10	Explain Bellman-Ford	34
8.11	Examenvragen	35
8.12	Oefeningen	35
8.12.1	Door het algoritme stappen	35
8.12.2	Categoriseren en argumenteren	36
<b>9</b>	<b>String Matching (Substring search)</b>	<b>36</b>
9.1	What is a Deterministic Finite state Automaton? (DFA)	36
9.2	Explain Knuth-Morris-Pratt (KMP)	36
9.3	Explain Boyer-Moore	36
9.4	Explain Rabin-Karp	37
9.5	Examenvragen	37
9.6	Oefeningen	38
9.6.1	KMP-toestandsmachine opstellen	38
<b>10</b>	<b>Dynamic Programming</b>	<b>38</b>
10.1	Explain Dynamic Programming	38
10.2	Explain Longest Common Subsequence (LCS)	38
10.3	Explain Optimal Binary Search Trees	39
10.4	Examenvragen	39
10.5	Oefeningen	40
10.5.1	LCS variant: palindroom	40
10.5.2	LCS variant: Shortest Common Supersequence	40
10.5.3	LCS variant: eigen regels	40

## 1 Inleiding

### 1.1 What is an algorithm?

A finite, deterministic, and effective problem-solving method, suitable for implementation as a computer program. They contain procedural knowledge (how to calculate) as opposed to declarative knowledge (true or false).

## 1.2 Why study algorithms?

1. Intellectual stimulation
2. To become a better programmer/software dev
3. To unlock the secrets of the universe (formula based science to algorithm based science)
4. Smart solutions are often faster than obvious solutions
5. Reasoning about algorithms tells us how an algorithm scales with size of input
6. Reduce a problem to another problem for which we already have a nice algorithm

## 1.3 Explain the convex hull algorithm

Goal: connect all the points in a 2D field on the outer edges so all other points are surrounded

Time needed: sort input +  $2 \times$  loop over all the data

Algorithm:

from left to right, connect all points  
if angle between 3 points  $> 180$  deg:  
 $\Rightarrow$  remove it and connect the other 2  
continue till right most  
do the same for the bottom from right to left

Sorting and finding convex hull are equivalent with relation to complexity

## 1.4 Why analyze algorithms?

1. Predict performance
2. Compare algorithms
3. Provide guarantees
4. Understand the theoretical basis of algorithms

Practical: avoid performance bugs

## 1.5 Explain: Framework for analysing algorithms?

Scientific method: Observe, Hypothesize, Predict, Verify, Validate

Principles: Experiments must be reproducible, Hypotheses must be falsifiable

## 1.6 What simplifications do we use when analyzing algorithms?

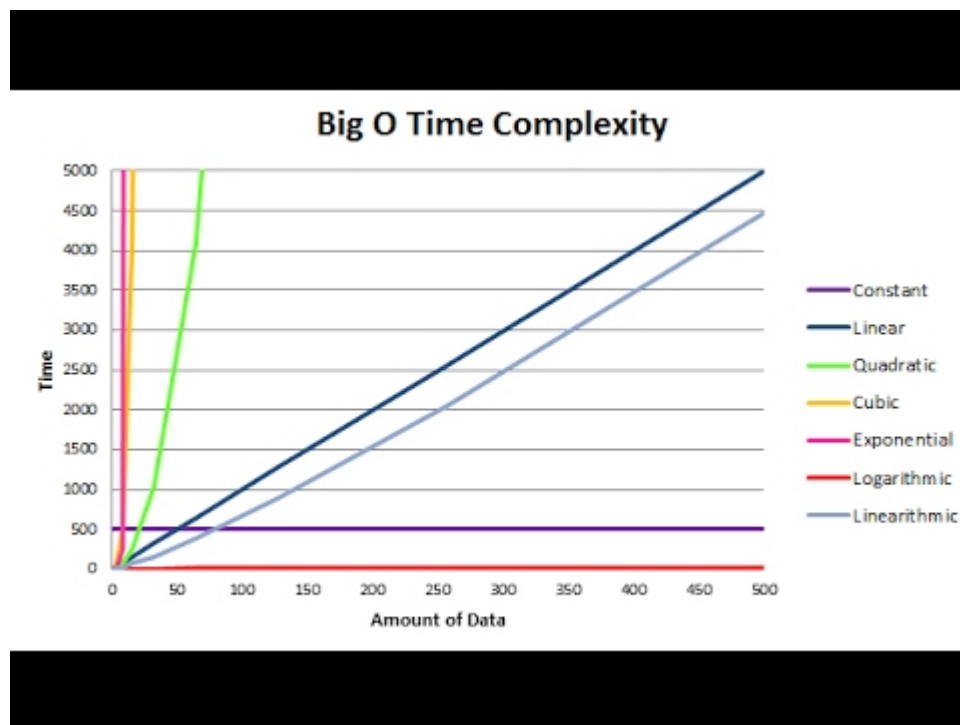
1. Count only a specific type of operation, proxy for all
2. Estimate running time (or memory) as function of input size  $N$  and ignore lower order terms

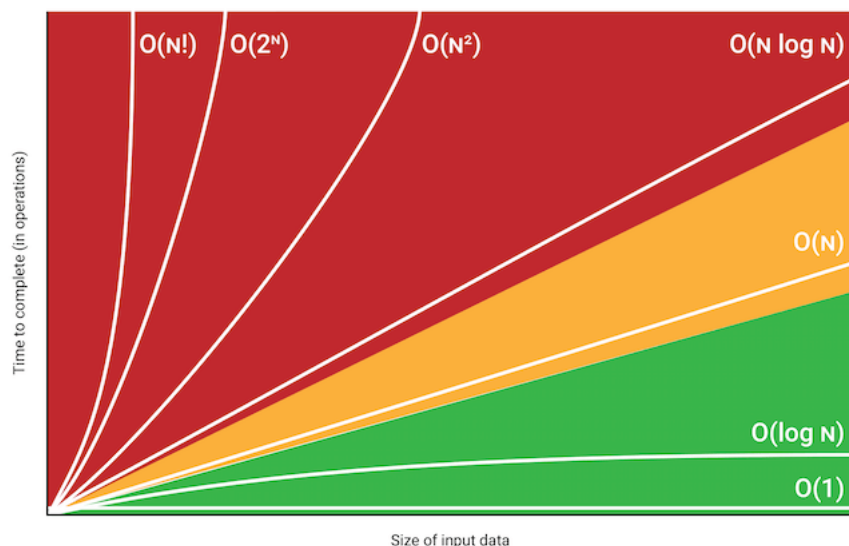
### 1.7 What are the differences between BigO and $\sim$ -notation?

1.  $O$  only provides an upper bounds, while  $\sim$  is simultaneously an upper and lower bound.
2.  $O$  only holds up to a constant  $f = O(g)$  if  $f(n) \leq C \times g(n)$  for some  $C > 0$ . In contrast, for  $\sim$  the implied constant is always 1: if  $f \sim g$  then  $\frac{f}{g} = 1$

## 1.8 Examenvragen

### 1.8.1 complexity charts





### 1.8.2 ~notation

We write  $\sim g(n)$  to represent any quantity that, when divided by  $f(n)$ , approaches 1 as  $n$  grows.

We also write  $g[n] \sim f(n)$  to indicate that  $\frac{g(n)}{f(n)}$  approaches 1 as  $n$  grows.

Example:  $\frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3}$  when divided by  $\frac{n^3}{6}$  approaches 1 as  $n$  grows, thus  $\sim \frac{n^3}{6}$

## 2 Sorting

### 2.1 Explain Selection Sort

---

#### Algorithm 1: Selection Sort

---

```

Sorted part at start of unsorted part;
while Unsorted part not empty do
    left to right;
    find smallest element;
    put it at the end of sorted part;
end

```

---

Performance:  $\sim \frac{n^2}{2}$  comparisons,  $\sim n$  exchanges

## 2.2 Explain Insertion Sort

---

**Algorithm 2:** Insertion Sort

---

Sorted part at start of unsorted part;  
**while** *Unsorted part not empty* **do**  
    left to right;  
    put new element in order in sorted part;  
**end**

---

Performance:

- Best if all elements of  $i$  are smaller:  $\sim n$
- Worst if all elements to left of  $i$  are larger:  $\sim \frac{n^2}{2}$
- Average if half of elements to left of  $i$  are larger:  $\sim \frac{n^2}{4}$

## 2.3 Explain Merge Sort

---

**Algorithm 3:** Merge Sort

---

Divide in two parts until only 2 elements;  
Sort those recursively;  
Merge them;

---

Performance:  $\sim n \log_2 n$  compares and data moves,  $\sim n$  extra space (aux memory)

Improvements:

- Switch to Insertion Sort for small arrays
- Only merge when necessary (compare last element of left and first of right)
- Use same aux array each time

## 2.4 Compare the two sorting algorithm types

- Comparison sorts sort by comparing two elements and need at least  $\sim n \log_2 n$  comparisons.
- Non-comparison sorts like counting sort, radix sort and bucket sort, use linear time ( $\sim n$ )

## 2.5 Explain: Benadering van Stirling

Een benadering voor de faculteit van grote getallen. Omzetten naar natuurlijk logaritme:

$$\begin{aligned}\log_2(n!) &= \log_2(e) \ln(n!) \\ &= \log_2(e) (\ln(n(n-1)(n-2) \dots 1)) \\ &= \log_2(e) (\ln(n) + \ln(n-1) + \ln(n-2) + \dots + \ln(1))\end{aligned}\tag{1}$$

Benadering van som naar integraal mogelijk, slechts kleine benaderingsfout.  
Met Simsombenadering

$$\ln(n) + \ln(n-1) + \ln(n-2) + \dots + \ln(1)$$

omvormen naar

$$\int_1^n \ln(x) dx \Rightarrow [x \ln(x) - x]_1^n \Rightarrow n \ln(n) - n + 1$$

$$\begin{aligned} \log_2(n!) &= n \log_2(n) - n \log_2(e) + \log_2(e) \\ &\Rightarrow \log_2(n!) \sim n \log_2(n) \end{aligned}$$

Dit is snel maar heeft dubbel zo veel geheugen nodig als de originele array.

## 2.6 Examenvragen

*Is de volgende uitspraak waar of niet waar? "Er bestaat een algoritme, gebaseerd op onderlinge vergelijkingen van elementen, dat 5 willekeurige elementen steeds kan sorteren door gebruik te maken van maximaal 6 vergelijkingen." Verklaar tevens je antwoord."*

Er kan voor een algoritme dat gebaseerd is op onderlinge vergelijkingen van elementen bewezen worden dat het niet sneller zal opgelost worden dan  $\sim \log_2 N!$ . Voor grote  $N$  zal deze formule door de Stirling-benadering geschreven kunnen worden als  $\sim N \log_2 N$ , maar voor dit geval is  $N$  klein. Ingevuld geeft de formule:

$$\log_2 N! = \log_2 5! = \log_2 120 \in [6, 7]$$

er zijn dus minstens 7 vergelijkingen nodig. De uitspraak is dus fout.

## 2.7 Oefeningen

### 2.7.1 Tabel lezen

*Gegeven één of andere tabel met probleemgroottes  $N$  en uitvoeringstijden (zie bvb. doubling experiment). Wat kan je besluiten ivm het gedrag van dit specifieke algoritme?*

Lineair/logaritmische/kwadratische tijdscomplexiteit

the order of growth is  $N^3$ ; to predict running times, multiply the last observed running time by  $2^b = 2^3 = 8$



### 2.7.2 Grafiek schetsen

*Stel dat ik voor een groot aantal verschillende arrays van verschillende lengte  $N$  zowel SelectionSort, InsertSort, als MergeSort laat lopen om de arrays te sorteren. Geef een grafiek van de uitvoeringstijden i.f.v.  $N$  (schets de grafiek, verklaar de grafiek, verklaar het verloop van de verschillende functies, ...) (Belangrijk: grafieken hebben geen zin als je niet minstens de assen benoemt, en effectieve getallen op de assen plot. Zeker voor vergelijkingen tussen functies moeten onderlinge verhoudingen juist zijn.)*

SelectionSort:  $\sim n^2/2$  comparisons,  $\sim n$  exchanges

InsertSort: Best if all elements to left of  $i$  are smaller:  $\sim n$ , Worst if all elements to left of  $i$  are larger:  $\sim n^2/2$ , Average if half of elements to left of  $i$  are larger:  $\sim n^2/4$

MergeSort:  $\sim n \log_2 n$  comp and data moves

## 3 QuickSort

### 3.1 Explain QuickSort

---

**Algorithm 4:** QuickSort

---

Take a pivot;  
Use the pivot to partition the list;  
Recursively call QuickSort on partitions;

---

Performance:

No additional memory needed

- Average:  $\sim 1.39n \log_2 n$  comparisons
- Worst case: already sorted array:  $\sim \frac{n^2}{2}$  chance:  $\frac{(2^{n-1})}{n!}$
- Best case: split is balanced:  $\sim n \log_2 n$

### 3.2 Give the first Partition variant

Use extra arrays for left and right, then concatenate (creating takes time)

---

**Algorithm 5:** Partition variant 1

---

```
Take a pivot;  
Create 2 arrays;  
forall Element e do  
    if  $e < pivot$  then  
        | put in array 1  
    end  
    if  $e > pivot$  then  
        | put in array 2  
    end  
    if  $e == pivot$  then  
        | put in either array  
    end  
end  
Concatenate: array1 + pivot + array2;
```

---

### 3.3 Give the second Partition variant

Exchange all elements bigger than pivot from left to right with elements smaller than pivot from right to left, then place pivot where the two loop iterators meet.

---

**Algorithm 6:** Partition variant 2

---

```
Take a pivot;
Hold Left and Right scan indices;
while True do
    // Scan right:
    while element < pivot do
        increase Left index;
        if Left index == arraySize then
            break;
        end
    end
    // Scan left:
    while element > pivot do
        increase Right index;
        if Right index == arrayStart then
            break;
        end
    end
    if Left index passed Right index then
        break;
    end
    Exchange elements in Right and Left index;
end
Put the pivot in position;
```

---

### 3.4 Give the third Partition variant: Lomuto

Has more exchanges but is more elegant

---

**Algorithm 7:** Lomuto partitioning

---

```
Set scan index as first element position;
Set the last element as pivot;
forall Element e do
    if  $e \leq pivot$  then
        Exchange element at scan index with position current element.
        Increase scan index;
    end
end
Put the pivot in position at scan index;
```

---

### 3.5 Give some optimizations

- Cutoff to Insertion Sort
- Median of 3 values for pivot: better probability of splitting roughly in half

- 3-way partitioning: when many similar values
- Random shuffle: against worst case

### 3.6 Examenvragen

*Bespreek de partitionering van Lomuto voor het partitioneren van een array bij het QuickSort algoritme. Geef een intuïtieve beschrijving, pseudocode, bespreek eventuele voor- en nadelen, alsook de tijdscomplexiteit.*

- Beschrijving:  
Plaats de pivot op de laatste positie. Neem het eerste element uit de deelrij met nog niet vergeleken elementen en vergelijk het met de pivot. Plaats dit element achter de linker deelrij als kleiner dan de pivot. Pas vervolgens de grenzen van de deelrijen aan en herhaal tot alle elementen in de juiste deelrij staan.

- Pseudocode:

---

**Algorithm 8:** Lomuto partitioning

---

```

Data: array A, lower index lo, upper index hi
pivot = A[hi] // pivot set as last element
i = lo // place for swapping
for  $j \leftarrow lo$  to  $hi-1$  do
    if  $A[j] \leq pivot$  then
        swap(A[i],A[j])
        i = i+1
    end
end
swap(A[i],A[hi])
return i

```

---

- Voordeel: eleganter
- Nadeel: meer verwisselingen
- Complexiteit:  
In dit algoritme wordt er tussen lo en hi-1 telkens 1 vergelijking gemaakt.  
Dit resulteert dus in een complexiteit van  $\sim N-1$

### 3.7 Oefeningen

#### 3.7.1 Bewijs dat de tijdscomplexiteit van Quicksort $1.39n \log n$ is.

*Bewijs.* Neem  $C_n$  het gemiddelde aantal vergelijkingen die nodig zijn om n verschillende elementen te sorteren.

We hebben  $C_0 = C_1 = 0$  en voor  $n > 1$  kunnen we de volgende relatie neerschrijven:

$$C_n = n + 1 + (C_0 + C_1 + \dots + C_{n-2} + C_{n-1})/n + (C_{n-1} + C_{n-2} + \dots + C_0)/n$$

Hierin zien we dezelfde structuur als het recursieve algoritme:

- de kost van partitioneren:  $\max(n + 1)$
- de gemiddelde kost van de linker subarray (kost alle elementen/aantal elementen)
- de gemiddelde kost van de rechter subarray (zelfde als die van rechter subarray)

Vermenigvuldig met  $n$  (deling wegwerken) en voeg gelijke kosten (linker en rechter subarray) samen:

$$nC_n = n(n + 1) + 2(C_0 + C_1 + \dots + C_{n-2} + C_{n-1})$$

Focus op 1 enkele stap, dus trek van  $nC_n$   $(n - 1)C_{n-1}$  af:

$$nC_n - (n - 1)C_{n-1} = 2n + 2C_{n-1}$$

Verwissel de termen:

$$nC_n = (n + 1)C_{n-1} + 2n$$

Deel door  $n(n + 1)$ :

$$\frac{C_n}{(n + 1)} = \frac{C_{n-1}}{n} + \frac{2}{(n + 1)}$$

Dan vinden we het resultaat:

$$C_n = 2(n + 1)\left(\frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{(n + 1)}\right)$$

We integreren dus om het oppervlak onder de  $\frac{1}{x}$  curve te vinden tussen 3 en  $n+1$ .

We vinden dan  $C_n \sim 2n \ln n \approx 1.39n \lg n$  □

### 3.7.2 Quicksort aanpassing

*Indien we Quicksort zouden implementeren met 2 pivots (en 3 partities), hoe zou de tijdscomplexiteit zich gedragen? Let op: dit is niet hetzelfde als een 3-way Quicksort waarbij 1 van de partities enkel elementen bevat die gelijk zijn aan de pivot.*

Voor elke recursieve stap zal ongeveer  $\frac{1}{3}$  van de elementen enkel met de laagste pivot vergeleken worden (de elementen die  $<$  zijn dan die pivot), de rest zal met beide pivots vergeleken worden. De kost van elke individuele stap zal dus iets hoger zijn ( $n(1 \times \frac{1}{3} + 2 \times \frac{2}{3})$  versus  $n$ ).

Het algoritme wordt dan 3 keer recursief opgeroepen, maar aangezien de subarrays kleiner zijn dan bij gewone quicksort (gemiddeld  $\frac{n}{3}$  versus  $\frac{n}{2}$ ) zullen deze telkens sneller gesorteerd worden.

We hebben dus een iets hoger kost per stap, maar minder stappen nodig om te sorteren.

Aangezien we 2 pivots hebben en minder recursieve stappen nodig hebben om te sorteren, zal een slecht gekozen pivot dus minder negatieve invloed hebben op de complexiteit.

Deze aanpassing van QuickSort zal dus een betere average tijdscomplexiteit hebben en dezelfde worst-case tijdscomplexiteit.

### 3.7.3 Quicksort worst case

*De worst-case tijdscomplexiteit van QuickSort is  $\sim n^2$ , indien men telkens voor elke partitionering, het kleinste dan wel het grootste element als pivot kiest. Men kan echter stellen dat QuickSort vrij vergevingsgezind is, en dat, zelfs als we toevallig enkele malen een slechte pivot kiezen gedurende de hele sortering, we toch een lineair verloop van de rekentijd kunnen verwachten. Verklaar!*

1 partitioningsstap vraagt  $\sim n$  tijd, nadien zijn nog  $n-1$  elementen te sorteren.

Als alle andere pivots wel uniform gekozen worden is de totale rekentijd

$$1.39n \lg n + n$$

(laatste  $n = \max$  kost extra partitionering).

Als er  $x$  slechte pivots zijn wordt dit

$$1.39n \lg n + xn$$

(maximaal).

Als  $x$  een vast getal is en niet-lineair scaleert met  $n$ , hebben we nog steeds een lineair verloop.

Daarnaast is de kans dat dit gebeurt, als de pivots uniform gekozen zouden zijn, zeer klein.

QS is dus wel degelijk vergevingsgezind.

## 4 Sorteren in lineaire tijd

### 4.1 Explain Bucket Sort

---

#### Algorithm 9: Bucket Sort

---

Divide input range into  $k$  buckets;

Sort each bucket using comparison-based sort;

---

Performance:

- Worst case: everything in 1 bucket

- Best case:  $\frac{n}{k}$  elements in each bucket
- Average: assume uniform random input resulting in equal size buckets: same as best case

Only works well for evenly distributed inputs, requires  $\sim n$  extra space

## 4.2 How many buckets are needed and how are they sorted?

If we assume  $\sim cn \log_2(n)$  sorting algorithm in each bucket, then the average/best case is  $\sim cn \log_2(\frac{n}{k})$ :

- if  $k$  is constant:  $\sim cn \log_2(n)$
- if  $k$  is proportional to  $n$ :  $\log_2(\frac{n}{k})$  is constant  $c'$ , so  $\sim c'n$

Sometimes the buckets don't need to be sorted, otherwise:

- few elements: insertion sort
- many elements: quicksort or mergesort

## 4.3 Explain Key-Indexed (Counting) Sort

Idea: Sort based on a small number of keys, count how many times each key occurs

---

### Algorithm 10: Counting Sort

---

```

Loop to count frequencies //  $\sim N$ 
Loop to compute cumulates //  $\sim R$ 
Loop to sort //  $\sim N$ 
Loop to copy back //  $\sim N$ 

```

---

Performance: Uses  $8N + 3R + 1$  array accesses to sort  $N$  records whose keys are ints between 0 and  $R-1$

Uses extra space proportional to  $N+R$

## 4.4 What is an in-place sorting algorithm?

An in-place algorithm is an algorithm that does not need extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'. However, a small constant extra space used for variables is allowed.

## 4.5 What is a stable sorting algorithm?

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

## 4.6 Explain LSD (radix) sort

Least Significant Digit

---

**Algorithm 11:** Radix Sort

---

```
for  $i \leftarrow 0$  to  $length$  do
  | Stable counting sort using  $dth$  ( $length - i$ ) character as key;
end
```

---

Performance:  $\sim 7WN + 3WR$  array accesses with  $W = length$  and  $\sim N+R$  extra space

Good for strings of equal length

Requires stable sort to be correct: if equal characters are sorted an unstable sort doesn't always give the same result and thus the overarching LSD sort could be wrong.

## 4.7 Explain MSD sort

Most Significant Digit

---

**Algorithm 12:** MSD Sort

---

```
Stable counting sort using first character as key;
Recursively apply counting sort to following characters;
```

---

Performance:  $\sim N \log_R N$  character comparisons (average:  $N$  strings,  $R$  character alphabet)

Good for strings of varying length

R-way recursion tree: many invocations of counting sort

Cut-off to insertion sort

Examines just enough characters to sort

Can be sublinear

## 4.8 Explain 3-way string QuickSort

Replace counting sort in MSD with quicksort, eliminates dependency on  $R$

---

**Algorithm 13:** 3-way QuickSort

---

```
Partition into  $<$ ,  $=$  and  $>$  pivot;
Recursively sort same char on  $<$  and  $>$  arrays;
Recursively sort next char of  $=$  array;
```

---

Performance: Uses  $1.39N \log_2 N$  character compares on average

Avoids recomparing initial parts of the string

Adapts to data (just enough compares)

Sublinear when strings are long

Good for strings because good chance of same character



#### 4.9 What are some key lessons from sorting?

- Many approaches for sorting are possible
- Many seemingly distinct problems are closely related
- Use reductions: transform one problem into another problem, to reason about algorithms
- Divide-and-conquer

#### 4.10 Kan een niet-stabiel sorteeralgoritme stabiel gemaakt worden?

Ja, meestal kan dit bereikt worden door de sleutelvergelijkings-algoritme aan te passen zodat de vergelijking van 2 sleutels de positie als factor beschouwt voor objecten met dezelfde sleutels.

#### 4.11 Examenvragen

*Besprek het CountingSort algoritme. Geef een intuïtieve beschrijving, pseudocode, en een analyse van de tijdscomplexiteit.*

- Beschrijving:  
Voor elk soort element in een lijst zal een telling bijgehouden worden met hoe vaak dat soort element voorkomt. De lijst kan vervolgens gesorteerd gereconstrueerd worden aan de hand van de tellijst, door voor elke sleutel in de tellijst  $n$  elementen aan de gesorteerde lijst toe te voegen (met  $n$  de waarde van de sleutel).
- Pseudocode:

---

**Algorithm 14:** Counting Sort

---

```
Data: list A
N = length of A
aux = new list of length N
count = new list of length R // R = number of different elements
      in A
for  $i \leftarrow 0$  to  $N-1$  do
  | count[A[i]+1]++ // count frequencies
end
for  $i \leftarrow 0$  to  $R-1$  do
  | count[i+1] += count[i] // count cumulates
end
for  $i \leftarrow 0$  to  $N-1$  do
  | aux[count[A[i]]+] = A[i] // sort: voeg op index countValue
  | countKey toe aan aux en doe countValue+1
end
for  $i \leftarrow 0$  to  $N-1$  do
  | A[i] = aux[i] // copy back
end
```

---

- Complexiteit:

In de eerste lus worden  $2N$  array accesses gebruikt, in de tweede  $3R$ , in de derde  $4N$  en in de vierde  $2N$ . Samen geeft dit  $\sim 8N + 3R$ .

***Wat betekent het als een sorteeralgoritme "stabiel" wordt genoemd? Welk sorteeralgoritme maakt hier expliciet gebruik van? Verklaar.***

Een stabiel algoritme is een algoritme dat de structuur die in de data zit behoudt. Een voorbeeld hiervan is indien gekeken wordt naar een lijst van personen die al alfabetisch gesorteerd is en men gaat deze opnieuw sorteren met een stabiel algoritme op leeftijd. Dan zal binnen elke leeftijdklasse de alfabetische volgorde behouden zijn.

LSD maakt gebruik van stabiliteit om te werken. Het gebruikt het stabiele CountingSort. Een lijst zal gesorteerd worden door te beginnen bij het laatste character en dit te sorteren. Hierna het voorlaatste, enzovoort.

De stabiliteit zorgt er voor dat binnen elk character terug een gesorteerde lijst staat. Als dit er niet was zou er ook veel nutteloos werk gedaan zijn. ***Bespreek het Least-Significant-Digit sorteeralgoritme. Geef een intuïtieve beschrijving, pseudocode en een analyse van de tijdscomplexiteit.***

- Beschrijving:

LSD is een algoritme dat gebaseerd is op de stabiliteitseigenschap van andere algoritmes. In de meeste implementaties zal gebruik gemaakt worden van CountingSort omdat dit algoritme deze eigenschap heeft.

De exacte uitwerking van LSD bestaat erin dat men eerst op de minst belangrijke digit zal sorteren. Vervolgens zal het voorlaatste karakter

gesorteerd worden en zo verder tot en met het eerste karakter.

- Pseudocode:

---

**Algorithm 15: LSD**

---

**Data:** list A  
W = length of elements in A **for**  $i \leftarrow W - 1$  **to** 0 **do**  
| CountingSort(A,i)  
**end**

---

- Complexiteit:

CountingSort word W keer opgeroepen, dus W keer  $\sim 8N + 3R \Rightarrow \sim 8WN + 3WR$

*Becommentarieer de volgende uitspraak: "HeapSort kan gebruikt worden ipv Counting Sort, als de basis voor LSD sortering".*

LSD is gebaseerd op de stabiliteit van het onderliggend sorteeralgoritme. Stabiele algoritmes behouden voorgaande structuren.

HeapSort is echter niet stabiel en kan dus niet gebruikt worden tenzij men HeapSort stabiel maakt door de oorspronkelijke positie van de elementen in rekening te brengen bij het sorteren.

Dit kost extra tijd, daarbij heeft HeapSort reeds een hogere complexiteit ( $\sim N \log_2 N$ ) dan CountingSort ( $\sim 8N + 3R$ ).

Er is dus geen voordeel bij het gebruiken van HeapSort over CountingSort, naast het iets lagere geheugengebruik.

## 4.12 Oefeningen

### 4.12.1 Counting sort aanpassing

*Hoe zou je counting sort kunnen wijzigen zodat het ook zou werken met data die geen positieve integers zijn? Bvb ook negatieve integers behandelen? Of ook getallen met decimale fracties? Welke voorwaarden moeten er gelden voor de te sorteren data?*

Aparte key array voor negatieve nummers.

Fracties wegwerken, sort, fracties terugbrengen of recursief: tel eerste digit pos i, dan tussen 2 digits pos i en i+1 2de digit j tellen en i+j

De data moet eindig en vergelijkbaar zijn.

## 5 Stacks,Queues & Hash Tabellen

### 5.1 Explain a stack

A container that removes the most recently added item.

## 5.2 What are the differences between a stack implemented with linked list vs array?

- Linked list pops and pushes in constant time and grows linearly in memory, it uses extra time and space to deal with the links.
- Array will need resizing, every operation takes constant amortized time (considers best and worst case) and there's less wasted space

## 5.3 Explain a queue

A container that removes the oldest item.

## 5.4 Explain hash tables

A container that maps the input based on the hashed key values.

The hash function should:

- be easy to compute
- result in a uniform distribution
- work so that the same keys result in the same hashes

Often primes are used as modulus as they result in less collisions.

Collisions are handled by different methods, such as separate chaining and linear probing.

## 5.5 What is Separate Chaining?

In this method an array of linked lists represents the hash table.

- Multiple values can be stored at every key position by simply adding them to the respective linked list.
- The number of probes for search/insert is  $\sim N/M$  (with  $M$  number of keys).
- Resizing is possible by rehashing, this is a way of reaching a constant average length of lists of  $N/M$ .
- Delete operation is easy to implement.
- Performance degrades gracefully
- Clustering is less sensitive to badly-designed hash functions

## 5.6 What is Linear Probing?

Store  $N$  key-value pairs in a table of size  $M$ , with  $M > N$ . The empty slots are used to resolve collisions; if a slot is taken, use the next available slot.

- Searching goes through the table linearly starting at the key, until the element is found or an empty position is reached.
- Cost depends on clusters of elements in table, longer clusters tend to become longer.
- Resizing requires all keys to be rehashed as the array size gets doubled or halved.
- Deleting involves marking the entry with a tombstone, otherwise searching will fail.
- This method wastes less space but has a problem with clustering.

## 5.7 Examenvragen

*Besprek verschillende strategieën om collisions in hash-tabellen op te lossen. Geef duidelijk voor- en nadelen aan, en illustreer met relevante voorbeelden indien nodig.*

- Separate chaining:  
Posities worden als een linked list bekeken, er kunnen dan meerdere objecten op dezelfde plaats staan
  - Insert: voeg het object op de juiste positie toe als het laatste element in de gelinkte lijst op die positie.
  - Search: ga rechtstreeks naar de juiste positie, bekijk dan de lijst op die positie, zoektijd nagenoeg constant
  - Delete: zoek, verwijder dan het object in de lijst en pas de links aan
  - Voordelen: Simpel te implementeren, Hash-tabel kan eindeloos groeien
  - Nadelen: De linken behandelen kost tijd, sommige posities worden mogelijk niet gebruikt
- Linear probing:  
Als de positie bezet is, ga dan naar de eerstvolgende vrije positie, er kan maar 1 element per positie staan
  - Insert: probeer het element op de juiste plaats toe te voegen, indien dit niet lukt (= niet leeg en geen grafsteen): probeer het dan bij de volgende positie
  - Search: kijk naar de juiste positie, is dit het gezochte element, kijk dan naar de volgende tot succes of een lege positie

- Delete: zoek het element en vervang het met een grafsteen
- Voordelen: Minder geheugen gebruik want geen linken
- Nadelen: Moeilijkere implementatie, Last van ophoping

## 5.8 Oefeningen

### 5.8.1 Hashtabel

*Klopt de inhoud met de gegeven hashfunctie?*

...

## 6 Priority Queues & Balanced Trees

### 6.1 Explain Priority Queues

Priority Queues return the largest/smallest element. There's a choice between an ordered and unordered implementation;

- Unordered has constant insert cost and linear deletion or max cost
- Ordered has linear cost but constant deletion/max cost. The goal is  $\log N$  for each operation. ( $\sim n \log_2 n$ )

### 6.2 What is a Binary Tree?

Binary tree is either empty or a node with links to left and right binary trees.

A complete tree is perfectly balanced except for the bottom level.

The height of a tree with  $N$  nodes is  $1 + \log_2(N)$

### 6.3 Explain Binary Heap (max/min heap)

A heap-ordered binary tree, has keys in the nodes, the keys are not smaller than the children's keys.

- The array representation lists all the keys level by level, the parent of node  $k$  is at  $k/2$  and the children of node  $k$  are at  $2k$  and  $2k + 1$ .
- Promotion: (key is replaced by key that is larger than parent key) exchange key in node with key in parent, repeat until order is restored.
- Insertion: Add a node at the end, then swim it up, max  $\log_2 N$  compares.
- Demotion: Exchange key in node with key in largest child, repeat until heap order is restored, max  $2 \log_2 N$  compares.
- Delete max: Exchange root with last node, then sink down, at most  $2 \log_2 N$  compares.

- Build up: Insert each element in turn ( $\sim N \log_2 N$ ) or if all elements are already in array:
  1. Loop over all elements from node  $N/2$  (last parent) to node 1
  2. Sink the value in that node
  3. Heap is then constructed bottom-up,  $2N$  compares
- Overview: Insert and delete max in  $\log N$  time, max in constant time

## 6.4 Explain Heapsort

Sorting with the help of a heap:

1. Place all  $n$  elements in priority queue.
2. Remove the largest element and put it at the end of a container, and rearrange the queue, repeat  $n$  times.

The time complexity is the sum of building the heap and extracting the elements 1-by-1

Not stable

## 6.5 Explain Binary Search Trees (BST)

A binary tree in symmetric order, it is either empty or two disjoint binary trees (left and right).

Symmetric order: each node has a key, and every node's key is larger than all keys in its left subtree and smaller than all keys in its right subtree.

Many BSTs are possible for the same set of keys

- Search: return value or null, Iterative or recursive implementation, # compares = depth of node, average:  $\sim 1.39 \lg N$
- Insert: search for key, if found reset value, if not add new node, same # compares as search
- Find min: move left as far as possible, find max: move right as far as possible
- Delete:
  - Simple: leave tombstone
  - Min: go left until node with null on the left child, replace node by right child, update subtree counts
  - Hibbard: Search for node with key to be deleted:
    - \* no children: replace with null

- \* 1 child: replace with child
- \* 2 children: replace with the min of the right child, remove that min from right child subtree

Average cost of  $\sqrt{N}$

- BST has poor worst case performance (traverse all nodes)

## 6.6 Explain 2-3 Trees

A theoretical fix to keep a tree balanced, also known as B-tree.

Allow 1 or 2 keys per node:

- 2-node has 1 key and 2 children
- 3-node has 2 keys and 3 children

Every path from root to null link has the same length.

- Search: similar to binary: if between the two values in 3-node: go middle child
- Insert:
  - Insert into 2-node: replace 2-node with 3-node containing new element
  - Insert into 3-node whose parent is 2-node:
    1. replace 3-node with temporary 4-node containing new element
    2. replace parent 2-node with new 3-node containing middle key of 4-node child
    3. split remainder of 4-node into two 2-nodes
  - Insert into 3-node whose parent is 3-node:
    - \* replace 3-node with temporary 4-node containing new element
    - \* replace parent 3-node with new 4-node containing middle key of 4-node child
    - \* split remainder of child 4-node into two 2-nodes
    - \* recursively handle 4-node parent
  - If you reach the root and it's a 4-node: split the root into three 2-nodes, this will increase the height of the tree by 1
- Each transformation maintains perfect balance and symmetric order
- Worst case:  $\log_2 N$  (all 2-nodes)
- Best case:  $\log_3 N$  (all 3-nodes)
- Guaranteed logarithmic performance for search and insert



## 6.7 Explain Red-Black Trees

A practical fix to keep a tree balanced, represent 2-3 tree as BST, uses internal left-leaning links as 'glue' for 3-nodes.

Invariants:

1. No node has 2 red links connected to it
  2. Every path from root to null link has the same number of black links
  3. Red links lean left
- Implementation: 1 field in node indicates whether left link is red
  - Rotations and color flips are used to implement the theoretical changes in 2-3 nodes into the practical BST
  - Rotation:
    - Left rotation: orient a (temp) right red link to the left
    - Right rotation: orient a left red link (temp) to the right; also flips who is parent in BST and exchanges middle child
  - Color flip: Recolor to split a (temp) 4-node
  - Insert:
    - Basic: Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black tree operations
    - Insert tree with 1 node:
      - \* Left: red link to new node containing new element, red link to the left.
      - \* Right: attach new node with red link, rotate to make legal 3-node.
    - Insert into 2-node at bottom: Do standard BST insert, color new link red, if new red link is right link, rotate left
    - Insert into 3-node: Do standard BST insert, color new link red, rotate to balance 4-node if needed, flip colors to pass red link up one level, rotate to make lean left if needed.
  - Operation cost is guaranteed at  $2 \lg N$  with an average of  $\lg N$

## 6.8 Examenvragen

*Besprek de insert en delete-maximum operaties in een heap structuur.*

Een max-heap-structuur is een boom waarvan in elk punt uitgegaan wordt dat beide kinderen kleiner zijn dan het huidige punt.

Een heap is ook een complete boom, dit wil zeggen dat enkel de onderste diepte niet volledig opgevuld is en elk element op de laagste diepte helemaal links is gedrukt.

- Insert:

1. het element wordt helemaal onderaan links in de boom gestoken
2. het element zwemt omhoog tot het in de correcte positie staat:
  - (a) het element wordt met z'n parent vergeleken
  - (b) indien het element groter is dan de parent: wissel ze van plaats

De complexiteit van deze operatie is  $\sim \log_2 N$  aangezien er nooit meer vergelijkingen nodig zullen zijn dan helemaal naar boven te zwemmen.  $\log_2 N$  is namelijk de diepte van de boom.

- Delete-maximum:

1. Het root element is de max, dus dit wordt verwijderd.
2. Het kleinste element wordt dan in de root geplaatst.
3. De nieuwe root zinkt naar beneden tot het op de juiste plaats staat:
  - (a) Het element wordt vergeleken met z'n 2 kinderen.
  - (b) Als het kleiner is dan 1 van de 2 worden deze 2 van plaats gewisseld

Complexiteit:  $\sim 2 \log_2 N$  want elk element heeft 2 kinderen

***Stel dat je een groot aantal gegevens over personenwagens wil bijhouden in een database, waarbij je wagens wil opzoeken adhv hun nummerplaat. We willen enkel gegevens opzoeken, dus niets toevoegen of verwijderen. Bovendien past de volledige database in het geheugen van de computer.***

***Voor welke datastructuur uit de onderstaande lijst zou je opteren, en waarom? Van welke factoren hangt de keuze eventueel af?***

***Gesorteerde array - Gelinkte lijst, - Binaire zoekboom - Queue - Hashtabel***

- Gesorteerde array, queue en gelinkte lijst: hebben een lineaire zoektijd en zijn dus te traag om gewenst te zijn.
- Binaire zoekboom: heeft logaritmische zoektijden, we kunnen beter. Indien het geheugen een factor wordt bij snellere structuren kan deze wel gewenst worden.
- Hashtabel: heeft constante zoektijd aangezien de nummerplaten als unieke sleutels gebruikt kunnen worden. Dit is ideaal maar vraagt wel extra geheugen. Zowel bij linear probing als separate chaining kunnen er lege elementen geheugen innemen.

## 6.9 Oefeningen

### 6.9.1 Ternaire heap voor HeapSort

*Stel dat we ternaire heaps (t.t.z. een heap met 3 kinderen per knoop) zouden gebruiken voor HeapSort. Op welke manier zou dit de tijdscomplexiteit beïnvloeden?*

Elke zink-stap zal 3 vergelijkingen en 1 swap nodig hebben ipv 2 vergelijkingen en 1 swap in binaire heap.

Twee niveaus in een ternaire heap zullen  $3^2 = 9$  elementen bevatten, met dezelfde hoeveelheid vergelijkingen kunnen we in een binaire heap  $2^3 = 8$  elementen bereiken. We winnen dus een extra element.

Er zullen dus een constant aantal minder swaps en vergelijkingen nodig zijn.

### 6.9.2 Heap zonder boomstructuur

#### *Young-tableau*

Dit is een mooie illustratie hoe heaps niet noodzakelijk in boomstructuren moeten voorgesteld worden, maar bvb. ook een matrix-vorm mogelijk is.

Semistandaard Young-tableau:

- in elke rij is elk getal kleiner dan of gelijk aan het getal rechts ervan
- in elke kolom is elk getal kleiner dan het getal eronder, als dat bestaat

### 6.9.3 Binaire Heap

#### *Verklaar binaire heap.*

Een heap met de vorm van een binaire boom, elk element heeft 2 kinderen. Operaties: zie Examenvragen

## 7 Greedy Algoritmen

### 7.1 What is a greedy algorithm?

Greedy algorithm is an algorithm that, step-by-step, decides the optimal next step, based on the previous step. It's not necessarily globally optimal, but is pretty good with limited info.

### 7.2 Explain run-length encoding

Use 4-bit counts to represent alternating runs of 0s and 1s

### 7.3 Explain Huffman coding

Assign every letter a leaf in a binary tree, with 0 = left and 1 = right every letter will then get a prefix code to find it again in the tree.

If each letter has the best code, the entire text has the shortest possible encoding.

This prefix tree can be constructed by greedily linking two least frequent characters, the combined characters form a "new character" with as frequency the sum of their frequencies.

This implementation requires a priority queue:  $\sim n \log_2 n$

The code is valid only for a specific message and should be sent with the message.

## 7.4 Prove Huffmans optimality

- Lemma 1: Any optimal code tree is complete.
  - Indien de optimale boom niet uit een complete boom bestaat zou het mogelijk zijn om de boom wel compleet te maken door het element dat alleen staat omhoog te duwen. Dit resulteert in een optimalere code en dus is het lemma bewezen.
- Lemma 2: There exists an optimal subtree in which the two least frequent letters are siblings at the maximum depth.
  - Indien dit niet het geval was zijn er twee opties:
    - \* ze liggen op dezelfde diepte in de boom: er kan geschoven worden met elementen zodanig dat deze twee elementen wel siblings zijn.
    - \* een van de twee ligt hoger dan de andere: de verwisseling tussen de sibling van het laagste element en het hoger gelegen element een optimalere code geven. Dit is dus geen optie.

Induction: Proof for 2 chars, if we can construct optimal code for r-1 chars, we can for r chars.

- Assume we can compute the optimal code tree for fewer than r symbols (we have a lower limit since we know the optimal for 2 chars (small pyramid, each char is child of root))
- $T_{huff}$  is the code tree for a set of symbols and frequencies:  $(s_1, f_1), \dots, (s_r, f_r)$
- $s_i$  and  $s_j$  are the first two symbols chosen and have the lowest frequencies, they are replaced by symbol  $(s^*, f_i + f_j)$
- The algorithm then computes a tree  $T_{huff}^*$  with r-1 symbols
- $W(T_{huff}) = W(T_{huff}^*) + (f_i + f_j)$  with  $W$  = total number of bits in code

Consider the most optimal tree  $T$  for  $(s_1, f_1), \dots, (s_r, f_r)$ :

- $s_i$  and  $s_j$  must be at the deepest level and siblings (lemma 1 and 2)
- make a new tree  $T^*$  by merging  $s_i$  and  $s_j$  into their parent  $(s^*, f_i + f_j)$

- $W(T) = W(T^*) + (f_i + f_j)$

Induction:  $T_{huff}^*$  is optimal for  $r-1$  letters:  $W(T_{huff}^*) \leq W(T^*)$   
 So:

$$\begin{aligned} W(T_{huff}) &= W(T_{huff}^*) + (f_i + f_j) \\ &\leq W(T^*) + (f_i + f_j) \\ &\leq W(T) \end{aligned} \tag{2}$$

Hence  $T_{huff}$  is a most optimal tree for  $r$  letters

## 7.5 Examenvragen

### *Leg uit: Huffman coding*

Er wordt gezocht naar een efficiënte manier om aan datacompressie te doen. Gegeven een string maakt dit algoritme de optimale codering in de volgende stappen:

1. Tel elk karakter in de string.
2. Neem de 2 karakters die het minst voorkomen, deze vormen samen een tak van een boom, tel de frequenties op en beschouw de boom als een nieuw karakter met als frequentie deze som van 2 frequenties.
3. Herhaal tot elk element gebruikt is, dit maakt het greedy.
4. Elke splitsing naar links of rechts in de boom staat voor 0 of 1 in de codering. De codering is ook prefix-vrij.

Proof of optimality:

- Lemma 1: Any optimal code tree is complete.
  - Indien de optimale boom niet uit een complete boom bestaat zou het mogelijk zijn om de boom wel compleet te maken door het element dat alleen staat omhoog te duwen. Dit resulteert in een optimalere code en dus is het lemma bewezen.
- Lemma 2: There exists an optimal subtree in which the two least frequent letters are siblings at the maximum depth.
  - Indien dit niet het geval was zijn er twee opties:
    - \* ze liggen op dezelfde diepte in de boom: er kan geschoven worden met elementen zodanig dat deze twee elementen wel siblings zijn.
    - \* een van de twee ligt hoger dan de andere: de verwisseling tussen de sibling van het laagste element en het hoger gelegen element een optimalere code geven. Dit is dus geen optie.

Induction: Proof for 2 chars, if we can construct optimal code for r-1 chars, we can for r chars.

- Assume we can compute the optimal code tree for fewer than r symbols (we have a lower limit since we know the optimal for 2 chars (small pyramid, each char is child of root))
- $T_{huff}$  is the code tree for a set of symbols and frequencies:  $(s_1, f_1), \dots, (s_r, f_r)$
- $s_i$  and  $s_j$  are the first two symbols chosen and have the lowest frequencies, they are replaced by symbol  $(s^*, f_i + f_j)$
- The algorithm then computes a tree  $T_{huff}^*$  with r-1 symbols
- $W(T_{huff}) = W(T_{huff}^*) + (f_i + f_j)$  with  $W$  = total number of bits in code

Consider the most optimal tree  $T$  for  $(s_1, f_1), \dots, (s_r, f_r)$ :

- $s_i$  and  $s_j$  must be at the deepest level and siblings (lemma 1 and 2)
- make a new tree  $T^*$  by merging  $s_i$  and  $s_j$  into their parent  $(s^*, f_i + f_j)$
- $W(T) = W(T^*) + (f_i + f_j)$

Induction:  $T_{huff}^*$  is optimal for r-1 letters:  $W(T_{huff}^*) \leq W(T^*)$

So:

$$\begin{aligned} W(T_{huff}) &= W(T_{huff}^*) + (f_i + f_j) \\ &\leq W(T^*) + (f_i + f_j) \\ &\leq W(T) \end{aligned} \tag{3}$$

Hence  $T_{huff}$  is a most optimal tree for r letters

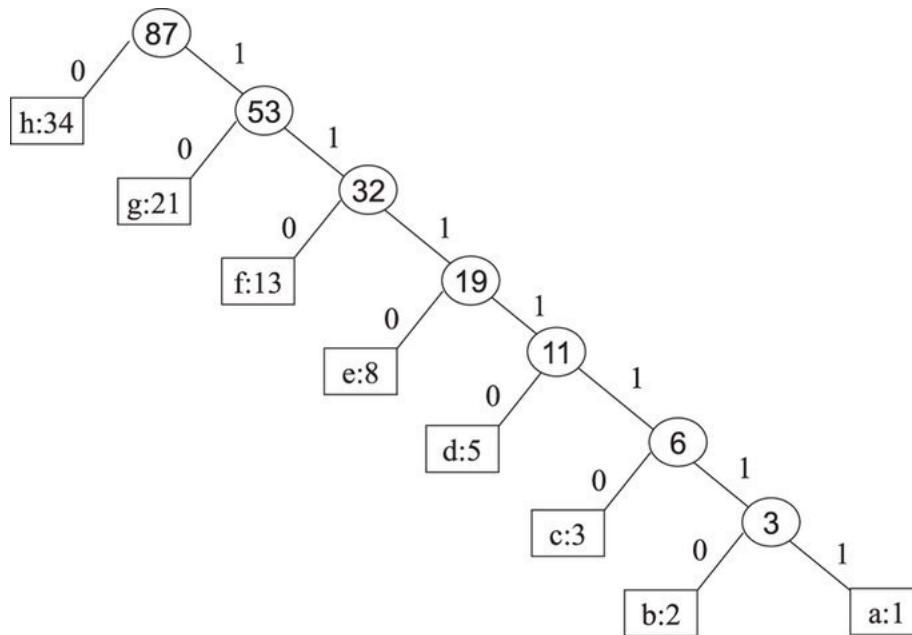
## 7.6 Oefeningen

### 7.6.1 Stel Huffman coderingsboom op

*Gegeven één of andere string, stel de Huffman coderingsboom op voor deze string.*

### 7.6.2 Huffman coderingsboom voor Fibonacci getallen

*Hoe ziet de Huffman codering er uit indien de frequentie van characters zich verhoudt tot de Fibonacci getallen?*



### 7.6.3 Aantal bits gecodeerde tekst

*Hoeveel bits zal een gecodeerde tekst (frequenties gegeven) van lengte  $n$  karakters tellen, als we gebruik maken van bovenstaande Huffman-codering?*

Aantal bits nodig per karakter wegen met de frequenties

### 7.6.4 Onder- en bovengrens afleiden

*Stel dat we een Huffman-codering opstellen voor een lengte  $n$  karakters, waarin slechts 3 verschillende karakters voorkomen. Kan je onder- en bovengrens afleiden voor het aantal bits (in functie van  $n$ ) dat de gecodeerde tekst zal tellen?*

3 chars: 1 char met lengte 1, 2 chars met lengte 2:

$$1 \times freq_1 + 2 \times freq_2 + 2 \times freq_3$$

met

$$freq_1 + freq_2 + freq_3 = 1$$

dan kunnen we ook schrijven  $2 - freq_1$ .

Aangezien  $freq_1 \geq freq_2$  en  $freq_1 \geq freq_3$  liggen de grenzen tussen  $n$  en  $1.66n$ .

Als  $freq_2$  en  $freq_3$  niet 0 dan kortst mogelijke lengte is  $n+2$ .

## 8 Minimum spanning tree and shortest path

### 8.1 Explain Minimum Spanning Tree (MST)

Given an undirected graph  $G$  with positive edge weights, find the spanning tree with minimum weight.

A spanning tree of  $G$  is a subgraph  $T$  that is connected and acyclic.

### 8.2 Explain the cut property

If the edge weights are distinct and the graph is connected, then a cut in a graph is a partition of its vertices into 2 (nonempty) sets.

A crossing edge connects a vertex in one set with a vertex in the other. The cut property states that given any cut, the crossing edge of min weight is in the MST.

Proof:

1. Let  $e$  be the min-weight crossing edge in cut, suppose  $e$  is not in MST, consider the graph of adding  $e$  to MST, this graph has a cycle that contains  $e$ , some other edge  $f$  in this cycle must be crossing edge of the cut.
2. Removing  $f$  and adding  $e$  is also a spanning tree
3. Since weight of  $e$  is less than the weight of  $f$ , that spanning tree has lower weight

Contradiction:  $e$  must belong to MST

### 8.3 Explain greedy MST algorithm

1. Start with all edges colored gray
2. Find a cut with no black crossing edges, color its min-weight edge black
3. Continue until  $V-1$  edges are colored black

Finding the cuts depends on the algorithm

### 8.4 Explain Prim's Algorithm

Start with vertex 0 and greedily grow tree  $T$ , at each step, add to  $T$  the min-weight edge with exactly one endpoint in  $T$

### 8.5 Explain Prim's Lazy implementation

Priority Queue of edge with (at least) one endpoint in  $T$

- Extract min to determine next edge  $e = vw$  to add to  $T$



- Disregard if both endpoint  $v$  and  $w$  are in  $T$
- Otherwise let  $v$  be the vertex not in  $T$ : add any edge incident to  $v$  to  $PQ$  and add  $v$  to  $T$

Removing obsolete edges in the middle would cost too much, which is why we only do it if they are the min

At most  $E$  edges in  $PQ$ : Insert and delete cost  $\sim E \log_2 E$ , with  $\sim E$  space needed for  $PQ$ , in practice  $PQ$  is much smaller than  $E$

## 8.6 Can you improve Prim's implementation?

Problem: obsolete edges are kept on  $PQ$

Fix:

- Maintain a  $PQ$  of vertices connected by an edge to  $T$
- Priority of vertex  $v$  = weight of shortest edge connecting  $v$  to  $T$
- Delete min vertex  $v$  and add associated edge  $e = vw$  to  $T$
- Update  $PQ$  by considering all edges  $e = vx$  incident to ( $=$  voortvloeiend uit)  $v$ :
  - $x$  already in  $T$ : ignore
  - $x$  not in  $PQ$ : add  $x$  to  $PQ$ , decrease priority of  $x$  if  $vx$  becomes shortest edge connecting  $x$  to  $T$

New running time:  $\sim E \log_2 V$ , space:  $\sim V$ , in practice for sparse graphs:  $E \sim V$

## 8.7 Explain Kruskal's algorithm

Consider edges in ascending order of weight, add the next edge to the tree  $T$  unless doing so would create a cycle

Special case of the greedy MST algorithm:

- Suppose Kruskal's algorithm selects edge  $e = vw$
- Cut = set of vertices connected to  $v$  (or to  $w$ ) in tree  $T$
- No crossing edge is black (no loops)
- No crossing edge has lower weight

Implementation uses a large queue, cycle detection is extra work: union find:

- Elk vertex heeft nummer van substructuur
- Als zelfde waard: zal lus vormen
- Anders: 1 van 2 substructuren nummer van andere geven

Time: Proportional to  $E \log_2 E$ : generally slower than Prim

## 8.8 Explain Shortest Path (SPT)

Find the shortest path from  $s$  to every other vertex, this can be represented in a tree with two vertex-indexed arrays

Edge relaxation: relax edge  $e = v \rightarrow w$ : if  $e$  gives a shorter path to  $w$  through  $v$ , update the shortest known path from  $s$  to  $w$  and last edge on shortest known path from  $s$  to  $w$

Generic algorithm:

1. Initialize  $\text{distTo}[s] = 0$  and  $\text{distTo}[v] = \infty$  for all  $v \neq s$
2. Repeat until optimal conditions are satisfied: relax any edge

No further relaxation is possible if  $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$

## 8.9 Explain Dijkstras algorithm

Almost identical to Prim

1. Consider vertices in increasing order of distance from  $s$  (non-SPT vertex with lowest  $\text{distTo}[]$  value)
2. Add vertex to SPT and relax all edges starting from that vertex

Proof:

- Each edge  $e$  is relaxed exactly once, leaving  $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$
- Inequality holds until algorithm terminates because:  $\text{distTo}[w]$  cannot increase or  $\text{distTo}[v]$  will not change (because we have non-negative weights &  $v$  is not relaxed again)
- Thus, upon termination, shortest-paths optimality conditions hold

Proportional to  $E \log_2 V$ : Priority queue of  $V$  vertices and every edge update requires a 'decrease key' operation in the queue

## 8.10 Explain Bellman-Ford

In each pass relax all edges, make  $V$  passes, if  $\text{distTo}[V]$  does not change during pass  $i$ , no need to relax any edge starting from  $v$  in pass  $i+1$

Time:  $\sim EV$

Detect negative cycles: after  $V-1$  passes if there are still vertices on the queue a negative cycle is detected ( $\Rightarrow$  no SPT exists)

## 8.11 Examenvragen

*Geef het algoritme van Prim om een minimaal opspannende boom van een grafe te berekenen, en bepsreek en verklaar de tijdscomplexiteit.*

Vanaf een gegeven knoop wordt er uitgebreid tot een MST gevonden is.

- Een knoop kan worden toegevoegd als hij rechtstreeks verbonden kan worden met een eerder verbonden knoop. De boom zal dus in elke stap volledig verbonden zijn.
- Om hiervoor de juiste edges te kiezen zullen de edges verbonden met knopen die al in de boom zitten op een queue gestoken worden, gesorteerd op kost.
- Nu zijn er 2 verschillende implementaties:
  - Lazy Prim: alle edges worden op de queue gegooit zonder lus controle, enkel wanneer we pop-en negeren we lus-edges.  
Extra ruimte:  $\sim E$ , tijdscomplexiteit:  $\sim E \log_2 E$
  - Eager Prim: elke iteratie worden edges uit de queue gefilterd die een lus vormen  $\Rightarrow$  kleinere queue  
Extra ruimte:  $\sim V$ , tijdscomplexiteit:  $\sim E \log_2 V$   
in de praktijk  $E \sim V$ , extra tijd nodig voor filteren

*Geef het algoritme van Kruskal om een minimaal opspannende boom van een grafe te berekenen, en bepsreek en verklaar de tijdscomplexiteit.*

PQ gevuld met alle vrije edges die gesorteerd zijn op hun gewicht

- Er zal een queue ontstaan met max  $E$  elementen
- Uit deze queue zullen stuk voor stuk edges gehaald worden waarvoor nagekeken wordt of ze aan de MST mogen toegevoegd worden: als ze geen lus zullen maken in de voorlopig opgebouwde boom

Dit geeft aanleiding tot het regelmatig gebruik van insert en extract min op de queue. Beide hebben een complexiteit van  $\sim \log_2 N$ , en dit voor  $E$  elementen met max  $E$  elementen op de queue  $\Rightarrow \sim E \log_2 E$

## 8.12 Oefeningen

### 8.12.1 Door het algoritme stappen

*Gegeven één of andere grafe, geef de volgorde waarin edges/vertices worden toegevoegd in Kruskal/Prim/Dijkstra/..., of geef de toestand van de priority queue bij Kruskal/Prim/Dijkstra nadat een specifieke vertex toegevoegd is aan de MST of SPT.*

### 8.12.2 Categoriseren en argumenteren

*Leg uit: algoritme van Prim/Kruskal/Dijkstra. Zou je dit algoritme beschouwen als een greedy algoritme (argumenteer), dan wel als een dynamisch algoritme (argumenteer), beide (argumenteer), of geen van beide (argumenteer).*

Greedy als het probeert een optimalisatie probleem op te lossen bij elke stap met de optimale keuze in elke stap.

- Prim: greedy: kies in elke stap de optimale edge
- Kruskal: greedy: kies in elke stap de optimale edge
- Dijkstra: greedy: kies de lokale optimale edge in elke stap (= kies de edge naar de dichtstbijzijnde node die nog niet in de boom zit)

## 9 String Matching (Substring search)

### 9.1 What is a Deterministic Finite state Automaton? (DFA)

DFA is abstract string-searching machine, it has a finite number of states (including start and halt), exactly 1 transition for each char in the alphabet, accept if sequence of transitions leads to halt state.

### 9.2 Explain Knuth-Morris-Pratt (KMP)

If we match the first 5 chars but mismatch the 6th, we do not need to back up the text pointer

DFA: state-number = number of chars in pattern that have been matched, length of longest prefix of pat[] that is at end of txt

if a mismatch: search for biggest state to go back to (= longest matched suffix) and go back to that state

- Needs precomputation to set up DFA
- when running at most  $N + M$  character accesses
- construction in time and space proportional to  $NM$
- linear-time worst-case guarantee

### 9.3 Explain Boyer-Moore

Scan characters in pattern from right to left, can skip  $M$  text chars when finding one not in the pattern (can skip less than  $M$  when finding char which belongs to patterns but on different pos)

Not backup-less

How much to skip: rightmost occurrence of character  $c$  in pattern, -1 for characters not in the pattern

Best:  $\sim N/M$  char compares, Worst:  $\sim MN$

## 9.4 Explain Rabin-Karp

Basic idea = modular hashing:

1. compute a hash of pattern chars 0 to  $M-1$  (choose hash modulo  $Q$  yourself)
2. for each  $i$ , compute a hash of text chars  $i$  to  $M+i-1$
3. if pattern hash = text substring hash: possible match

Basically brute-force check the hash and check the chars if match

Horner's method:

- add  $n$  digits at the start (with  $n$  length of pattern)
- linear-time method to evaluate degree- $M$  polynomial

The hash function can be updated in constant time

- Las Vegas variant: if hash is equal: check pattern: extremely likely to run in linear time (Worst case:  $MN$ )
- Monte Carlo variant: if hash is equal: assume pattern is equal: always runs in linear time, probability for mismatch =  $1/Q$

Extends to finding multiple patterns, but arithmetic is slower than char compares and Las Vegas requires backup

## 9.5 Examenvragen

### *Leg het Boyer-Moore algoritme voor substring matching uit*

Er wordt getracht na te gaan of een substring in een gegeven string zit. Dit zal gebeuren door achteraan de substring te beginnen vergelijken.

3 opties:

1. Het element matched: probeer het element links van de match te matchen, is dit het eerste element, dan matched heel de substring.
2. Het element komt niet voor in de substring: kan nooit tot substring behoren, skip voor heel de substring: verplaats de substring tot na dit element en zoek terug vanaf het laatste element.
3. Geen match maar komt wel in de substring voor: er zal naar de meest rechtse keer dat dit karakter voorkomt gekeken moeten worden:
  - dit element is rechts van het huidige gecheckte element: een kleine verschuiving zal niet helpen, er wordt gedaan alsof dit element niet in de substring voorkomt

- dit element is links van het huidige gecheckte element: er kan mits een kleine verschuiving een geldige combinatie gevonden worden. De substring wordt verschoven tot de elementen matchen en er wordt opnieuw helemaal achteraan gekeken of het een match is.

## 9.6 Oefeningen

### 9.6.1 KMP-toestandsmachine opstellen

*Gegeven een string, stel de bijhorende Knuth-Morris-Pratt toestandsmachine op.*

## 10 Dynamic Programming

### 10.1 Explain Dynamic Programming

- Problem can be subdivided in subproblems
- optimal subsolutions result in optimal solution
- Store the result of sub...subproblems in a look-up table and re-use the results (=memoization)

By re-using the solutions of the sub...subproblems, the (time-)complexity for finding a solution can be significantly reduced

- Top-down storing: store when result is first encountered
- Bottom-up storing: start with subproblems and build up to larger problem

### 10.2 Explain Longest Common Subsequence (LCS)

Find out how alike the two sequences are  $\Rightarrow$  choose subsequences in each and find the longest matching subsequence between the two

A subsequence  $\neq$  substring: subsequence is a sequence of chars that follow each other but don't need to be neighbours

Solution:

- Naïve: find all subsequences of string 1 and see if they are subsequences of string 2, then return the longest match
- Better: given strings X (length n) and Y (length m):
  - length LCS of  $X[0 \dots i]$  and  $Y[0 \dots j] = L[i, j]$
  - if  $X[i] = Y[j]$ :  $L[i, j] = L[i-1, j-1] + 1$
  - if  $X[i] \neq Y[j]$ :  $L[i, j] = \max L[i-1, j], L[i, j-1]$

- Boundary cases:  $L[i,-1] = L[-1,j] = 0$

---

**Algorithm 16: LCS**


---

```

Fill out table L: starting at L[0,0] and working up, boundary cases are
not explicitly stored in the table;
Circle matching chars;
Count the LCS: ;
if same char then
|   upperleft + 1
else
|   max(left,up)
end
Follow the path from end to start;

```

---

Multiple solutions possible

Time complexity:  $\sim nm$

### 10.3 Explain Optimal Binary Search Trees

We want to build a BST with minimum expected searchcost:

- most prevalent values near the top
- but not necessarily the root (as it's still a BST so smaller values to left, bigger to right) (cost = number of compares)

We must construct the BST out of Optimal Substructures:

- Optimal left subtree has cost  $c(i,r-1)$
- Optimal right subtree has cost  $c(r+1,j)$

When combined the depth of each node in the subtrees increases by 1, the cost of the subtree increases by sum of all probabilities in subtree

We do not know the optimal  $r$  so we need to take the minimum over all possible  $r$ 's

### 10.4 Examenvragen

***Omschrijf het Longest Common Subsequence probleem zoals gezien in de les. Geef tevens een oplossing die gebruik maakt van dynamisch programmeren om LCS op te lossen.***

Men zoekt de langste subsequence die gedeeld wordt door een set sequences.

We lossen dit recursief op maar zorgen ervoor dat er geen dubbel werk verricht wordt. Maak een lookup table waarin eerder opgeloste bewerkingen opgeslagen worden.

Deze matrix van grootte  $M$  (lengte string 1)  $\times$   $N$  (lengte string 2) zal gevuld worden aan de hand van volgende rekenregels:

1.  $L[i][j] = L[i-1][j-1] + 1$  als  $X[i] = Y[j]$
2.  $L[i][j] = \max\{L[i-1][j], L[i][j-1]\}$  als  $X[i] \neq Y[j]$

Dit algoritme zal op deze manier een matrix vullen om op positie  $L[M][N]$  het antwoord op het probleem te vinden. Deze matrix opvullen heeft een complexiteit van  $\sim MN$

## 10.5 Oefeningen

### 10.5.1 LCS variant: palindroom

*Een variant op het LCS probleem is de langste subsequentie zoeken die tevens een palindroom is.*

Palindroom zoeken in 1 sequence: Zoek LCS voor sequence en reverse van sequence

### 10.5.2 LCS variant: Shortest Common Supersequence

*SCS of X and Y is the shortest sequence which has X and Y as subsequences*

1. Zoek de LCS van X en Y.
2. Voeg de niet-LCS characters (in hun originele volgorde) toe aan de LCS en return het resultaat.

Example 1:  $X = \text{"geek"}, Y = \text{"ëke"}$

1. LCS van X en Y:  $\text{"ëk"}$
2.  $\text{"ëk"}$  wordt  $\text{"geeke"}$ , dit is de SCS

Example 2:  $X = \text{"AGGTAB"}, Y = \text{"GXTXAYB"}$

1. LCS:  $\text{"GTAB"}$
2. SCS:  $\text{"AGXGTXAYB"}$

### 10.5.3 LCS variant: eigen regels

*Diverse rekenregels voor variant opstellen: als lengte 0 dan iets, als  $X(m) = Y(n)$  dan iets, als  $X(m) <> Y(n)$  dan iets*