1.4

Algoritmen: procedurale kennis (hoe dingen uit te voeren)

Convex hull = convex omhullende figuur
- kan niets maken buiten de figuur
- algoritme: binnenhoek < 180° → ok
  buitenhoek < 180° → probleem – punt overslaan, naar volgend punt
  zo eerst bovenkant punten verbinden, dan onderkant zelfde herhalen
  door alle punten op gesorteerde manier te overlopen (2x)
- sorteren en convex hull zijn "equivalent" m.b.t. hun complexiteit
  meest efficiënte sorteeralgoritme ⟺ meest efficiënte convex hull algoritme
  $T_{convex}(n) = T_{sort}(n) + cn + cn$
  bovenkant + onderkant doorlopen

  ($T_{sort}(n) = T_{convex}(n) + c'n$
  onderkant convex omhullende uitlezen → gesorteerde lijst

Analyseren → performantie, vergelijken met andere, begrijpen tgv. theor. basis

N-lichamenprobleem
- slecht algoritme: brute force
  2 for loops: elk lichaam met elk ander vgl'en
  → N(N-1) berekeningen ⇒ $\sim n^2$
- beter: Barnes-Hut:
  clusters bekijken met hun resulterende kracht
  → $N \log_2 N$

Wetenschappelijke methode voor analyseren
- observeren [natural world]
- hypothese opstellen [model consistent met observaties]
- voorspellen [gebruik hypothese]
- verifiëren [adhv nieuwe observ.]
- valideren [herhalen totdat het klopt]
  ↳ experimenten reproduceerbaar
  & hypothese falsifieerbaar

opmeten uitvoertijd experimenten [stopwatch] plotten ifv de invoergrootte
log-log plot: T(N) ifv N → $\log_2(T(N))$ ifv $\log_2(N)$
stel $T = a \cdot N^b$ → $\log_2(T) = b \log_2(N) + c$, met $a = 2^c$
rechte hierdoor fitten en fitparam verifiëren

verdubbelingsexperiment [werkt enkel bij power laws]
ook toepasbaar voor $T(n) \sim a n^b \log(n)$
gok een macht b
run experimenten met telkens een verdubbelende grootte N
we kunnen nu b verifiëren
aangezien voor hypothese $T = aN^b$ → $T(2N) = a(2N)^b$
en dan in de verhouding $\frac{T(2N)}{T(N)} = \frac{a(2N)^b}{a(N)^b} = 2^b$ ⇒ $b = \log_2(\text{ratio } \frac{T(2N)}{T(N)})$

$\frac{T(2N)}{T(N)} = \frac{a(2N)^b \log_2(2N)}{a N^b \log_2(N)}$
$= 2^b \frac{\log_2(2N)}{\log_2(N)}$
$= 2^b \left(\frac{\log_2(2)}{\log_2(N)} + 1\right)$
$= 2^b (1 + 1/\log(N))$
$\approx 2^b$

eenmaal je zo b hebt gevonden, kan je a eruit schatten
door voor eenzelfde N het experiment enkele keren te runnen
en dan oplossen voor a: $a = \frac{T}{N^b}$ met N gekend, T gemeten & b berekend
⇒ hypothese $T = aN^b$ met berekende a en b

bij $T = a N^b$ is

- a afhankelijk v/h spec. systeem, bv snelheid computer, OS, software, schrijven
  → systeem afh effecten
- b onafhankelijk v/h spec systeem
  → systeem onafh effecten → bv algoritme, input data

wiskundige modellen
(kost van elke operatie) x (frequency v/d operatie) ⇒ totale tijd
└ tijdsintensieve manier
→ simplificaties 1) bekijken enkel de bewerkingen die herechte werk doen
niet de boekhouding e.d. └ proxy voor heel het algoritme
→ tellen enkel de body,
werk i/h binnenste v/d lus
2) bekijken enkel de meest stijgende term, in termen van N,
want we zijn enkel geïnteresseerd in gedrag bij zeer grote N
↳ orde van groei
⇒ tilde notatie

$f(x) \sim g(x)$: $\lim_{x \to \infty} \frac{f(x)}{g(x)} = 1$ → ongeveer gelijk aan → leidende term
willekeurigheid

andere notaties
- grote O : $f(x)$ is $O(g(x))$: $\exists x_0, x \geq x_0 : f(x) \leq c\, g(x)$ → bovengrens groeigedrag
- grote Θ → asymptotische groeigelijkheid : f is $\Theta(g)$: f is $O(g)$ en g is $O(f)$
  - O : Θ en kleiner → bovengrens
  - Ω : Θ en groter → ondergrens

bij tilde: vergelijken meest stijgende term ook meenemen
kan heel bel! zijn in efficiëntie algoritmes met zelfde grote O
vergelijken, want $\sim \frac{1}{10} n^2$ kan veel efficiënter zijn dan $\sim 50 n^3$

valkuils in doorgelijke analyse
- grote ipv coëff laten ook termen
- niet-dominante binnenste lus
- instruction time niet voor elke instructie gelijk
- systeem pc
- sterke afh. input
- versch. probleemparam., niet enkel N

worst case - best case - average case vergelijken

amortized analysis: $\frac{\text{totale kost alle operaties}}{\#\text{operaties}}$ → gem kost laag proberen te houden

geheugengebruik ook bel!
geheugen object = som geheugen alle instance var. (~4 bytes per instance) + geheugengebr. overhead object (~16 bytes)
int[] ~4n bytes, double[] ~8n, double[][] ~8mn
ref. objectklasse + vuilbak info + synchronisatie info

→ rearranging array of elements based on the keys of the items

SORTING: sorteren van data
sorteralgoritme kiezen op basis vd data
- hoe makkelijk zijn de objecten te verplaatsen/kopiëren (haaien)
- " " is het om objecten te vergelijken (nijlpaardicht)
⇒ 2 soorten sorteeralgoritmen - vergelijkingen methvel gedaan
- manipuleren (verplaatsen/kopiëren) methvel gedaan

in-place sorting
+ invariabel gedrag

Selection sort
minimum in de rij zoeken, dit selecteren en vooraan zetten
dan in resterende rij hetzelfde doen
dit blijven herhalen totdat heel de rij gesorteerd is (geen rest rij meer over)

```
public class SelectionSort {
    public static void sort(Comparable[] a ){
        int n = a.length
        for(int i=0 ; i<n ; i++) {
            int min = i;
            for(int j= i+1; j<n ; j++) {
                if (less(a[j], a[min]))
                    min=j ;}
            exchange(a , i, min); }
    }
}
```

less private: ipv < opdat we iets aanpasbaar hebben wat bv5 ook voor strings kan werken

↳ 2 for lussen: 1 om door de lijst te ~~blijven~~ lopen totdat alles gesorteerd is
1 om door de lijst te lopen en telkens het min te zoeken
steeds kleiner wordend

alle elementen links van i zijn steeds/reeds gesorteerd en zij dus niet opnieuw bekijken
inclusief i
alles rechts van i kan nooit kleiner zijn dan de elementen links ervan
invarianten vh algoritme, altijd waar na de lus 1 kan te doorlopen

[symm algoritme: max zoeken en vanachter zetten, van achter ivt voor werken]

```
    private static boolean less ( Comparable v, Comparable w ){
        return v.compareTo(w) < 0 ; }
    private static void exchange (Comp[] a, int i, int j ){
        Comparable t = a[i];
        a[i] = a[j];
        a[j] = t; }
```

= verwisselingen

analyse: # vergelijkingen & # verplaatsingen tellen
n elementen
op iteratie i: (n-i-1) vergelijkingen, 1 verplaatsing
→ # vergelijkingen voor alle iteraties

$$\sum_{i=0}^{n-1} (n-i-1) = (n-1) + (n-2) + \dots + 1 + 0 = \frac{n(n-1)}{2}$$

$$\Rightarrow \sim \frac{n^2}{2}$$

→ # verwisselingen

$$1 + 1 + \dots + 1 = n-1 \Rightarrow \sim n$$

Insertion Sort

in-place sorting + variabel qadraag

1e element vld ongesorteerde lijst vergelijken met zijn (al gesorteerde) voorgangers
dit herhalen totdat het 1e ongesorteerde element op de juiste plaats in de reeds gesorteerde lijst kan in = tussengeschoven
herhalen voor alle elementen in de (steeds korter wordende) ongesorteerde lijst

```
public class InsertionSort {
    public static void sort (Comparable[] a) {
        int n = a.length;
        for (int i = 1; i < n; i++) {
            for (int j = i; j > 0 && less (a[j], a[j-1]); j--)
                exchange (a, j, j-1); }
    }
}
```

element pakken
vergelijken met zijn linker-buurman
indien nodig wisselen als linker > rechts

↳ 2 for lussen: 1 om de nog te sorteren rij steeds kleiner te maken
1 om het ongesorteerde elementje steeds in de gesorteerde lijst in te voegen

invar { alle elementen links (inclusief) i zijn gesorteerd
alle elementen rechts van i zijn mag niet beluisteren

analyse / performantie

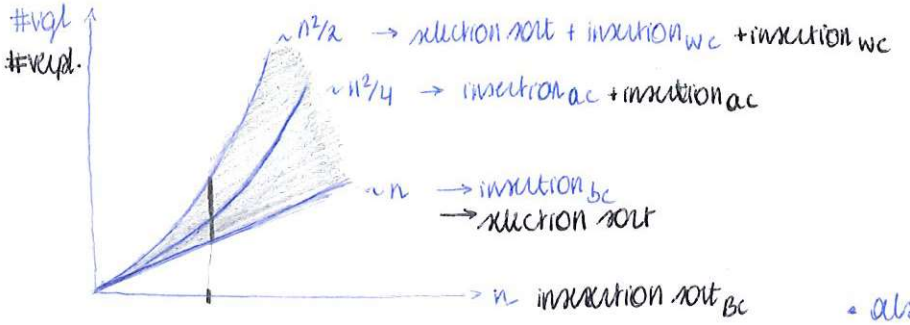
voor element i: per groter element links van i 1 vergelijking en 1 verwisseling
+ max 1 vergelijking ~~voor~~ met een kleiner element

best case: alle elementen links van i zijn kleiner
→ rij i/h begin al gesorteerd
0 verplaatsingen
n-1 vergelijkingen ~ n

worst case: alle elementen links van i zijn groter
→ rij i/h begin omgekeerd gesorteerd
$\sum_{i=0}^{n-1} (n-i-1) = \frac{n(n-1)}{2}$ vergelijkingen & verwisselingen ~ $\frac{n^2}{2}$

average case: helft vld elementen links van i zijn groter
→ elk element moet tot het middelen vld verplaatst w
⇒ # vergelijkingen & verplaatsingen ~ $\frac{n^2}{4}$

Selection vs Insertion Sort



qua # vgl'en is IS in het algemeen beter dan SS

qua # verplaatsingen is SS echter algemeen beter dan IS
↳ verhouding +/- als a afwegen

- als het vergelijken makkelijk is, vb int → selection sort
- als het vergelijken moeilijk is, vb strings → insertion sort

insertion sort is een heel goed algoritme als je rij reeds gesorteerd is,
vb als alle elementen max een vast getal x van hun juiste positie staan
dan lineair qadraag ipv kwadratisch, nl $(\frac{x}{2}+1)n$ # vgl'en
en $\frac{x}{2} n$ # verpl } average

bij SS werkt dit niet zo → blijven nog steeds ~$n^2/2$

bottom-up merge sort

```
public static void sort (Comparable[] a) {
    int N = a.length;
    aux = new Comparable[N];
    for (int sz = 1; sz < N; sz = sz+sz) {
        for (int lo = 0; lo < N-sz; lo = sz+sz) {
            merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
```

niet in-place sorting → extra geheugen nodig

Merge Sort

probleem opsplitsen in 2 delen - elk apart sorteren → $2 \times \sim (n/2)^2/2 = n^2/8 \rightarrow$ tot $\sim n^2/4$

→ 2 delen terug samenvoegen (mergen): kleinste vd 2 delen vergelijken en kleinste van die 2 in nieuwe rij zetten

⇒ extra geheugen nodig (extra hulparray: liefst dat later terug het oorspr. array)

└ recursieve oplossing + hulparray

```
public class MergeSort {
    private static Comparable[] aux;                    aux. array for merges
    public static void sort (Comparable[] a) {
        aux = new Comparable[a.length]                  allocate space just once
        sort (a, 0, a.length-1);
    }
    public static void sort (Comp[] a, int lo, int hi) {
        if (hi <= lo) return;                           → array of length 1 reached
        int mid = lo + (hi-lo)/2;
        sort (a, lo, mid);                              sort left half
        sort (a, mid+1, hi);                            sort right half
        merge (a, lo, mid, hi);                         merge
    }
```

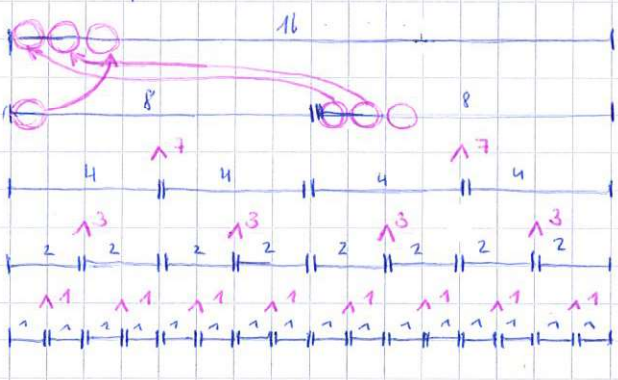
top-down merge sort

sort a[lo..hi]

merge a[lo..mid] with a[mid+1..hi]

```
public static void merge (Comp[] a, int lo, int mid, int hi) {
    for (int k = lo; k <= hi; k++)                      copy a[lo..hi] to aux[lo..hi]
        aux[k] = a[k];
    int i = lo, j = mid+1;                              → aux[j] = a2[0]
    for (int k = lo; k <= hi; k++) {                    merge back to a[lo..hi]
        if (i > mid)                                    if a1 is fully in a
            a[k] = aux[j++];                            → insert a2 in a
        else if (j > hi)                                if a2 is fully in a
            a[k] = aux[i++];                            → insert a1 in a
        else if (less(aux[j], aux[i]))                  if a2[0] < a1[0]
            a[k] = aux[j++];                            → put a2[0] in a
        else a[k] = aux[i++];                           if a1[0] <= a2[0] → put a1[0] in a
    }
```

aux[i] = a1[0]



voor $N = 2^n$ met $n = \#$ niv, hier 4

15 vgl'en = 1(2·8−1) — $N-1$

14 vgl'en = 2(2·4−1) — $2(\frac{N}{2}-1) = N-2$

12 vgl'en = 4(2·2−1) — $4(\frac{N}{4}-1) = N-4$

8 vgl'en = 8(2·1−1) — $8(\frac{N}{8}-1) = N-8$

~2k verplaatsingen en ~k vgl'en om 2 arrays van lengte k/2 te mergen

→ ~$n \log_2 n$ vergelijkingen en verplaatsingen [forum - pl 23]

~ n extra plaats nodig (geheugen hulparray)

worst case voor $N = 2^n$ (met n het # niveaus, N het # elementen)

$N \log_2 N - (1 + 2 + 4 + 8 + \dots) = N \log_2 N - (2^0 + 2^1 + 2^2 + \dots + 2^{n-1})$ ($2^n - 1$)

$= N \log_2 N - (N-1)$

$= N \log_2 N - N + 1$ exact

$\sim N \log_2 N$

best case $\sim \frac{N}{2} \log_2 N$ (een deling helemaal opgeteld veel groter/kleiner dan andere $\Rightarrow$ maar $N/2$ vgl'en nodig)

stel dat $N \neq 2^n$, dan gebruiken we dat $2^{m-1} < N < 2^m$ en ku we nog steeds een onder- en bovengrens bepalen

↳ worst & average case zijn sneller dan insertion sort (asymptotisch)
best case is insertion sort asymptotisch sneller dan merge sort

verbeteringen merge sort
- overschakelen op insertion sort voor kleine arrays [mergesort heeft te veel overhead voor kleine arrays]
- merge enkel wnnr nodig
  laatste element linkersubarray vergelijken met eerste element rechtersubarray
- gebruik steeds dezelfde subarray

Snelheid sorteeralgoritmen
- Comparison sorts:
  sorteren dr 2 elementen te vergelijken
  Minstens $\sim n \log_2 n$ vergelijkingen → merge sort is asympt. optimaal
- Non-comparison sorts
  vb counting sort, radix sort, bucket sort
  in lineair ($\sim n$) tijd

$\sim n \log_2 n$ als ondergrens voor comparison sorts

"B" ahv beslissingsboom: elke mogelijke input is een blad in de boom
$\Rightarrow n!$ verschillende bladeren in de boom

elke knoop heeft 2 uitkomstmogelijkheden (binair)
- 2 elementen die vergeleken worden

aangezien elke binaire boom met een hoogte h $\leq 2^h$ bladeren heeft
$\Rightarrow n! \leq$ # bladeren $< 2^h \Rightarrow h \geq \log_2(n!)$
→ boom telkens delen tot je tot 1 komt
$\Rightarrow \log_2(n!)$ niveaus

aangezien h (de min. lengte vd beslissingsboom) het worstcase # vgl'en is
kunnen we nu uit Stirlings benadering afleiden dat $\log_2(n!) \sim n \log_2 n$

$\Rightarrow$ geen enkel sorteeralgoritme (voor comparison sorts) kan worst case beter dan $\sim n \log_2 n$
→ merge sort asympt. optimaal [maar heeft wel extra geheugen nodig]

echte ondergrens is $\lceil \log_2(n!) \rceil$

bewijs $\log_2(n!) = \log_2 e \cdot \log_e(n!)$

$\ln(n \cdot (n-1)(n-2) \dots 2 \cdot 1)$
$= \ln(n) + \ln(n-1) + \dots + \ln(1)$
$\approx \int_1^n \ln x \, dx$ (integraalbenadering; andere benadering zou telkens voor extra lagere orde termen zorgen)
$= [x \ln x - x]_1^n = n \ln(n) - n + 1$

$\rightarrow \log_2(n!) \approx \log_2 e \, (n \ln(n) - n + 1)$
$= n \log_2(n) - 1.4427n + 1.4427$ $\sim n \log_2 n$

(stel ~ completed sort
stap ~ comparison tussen a[i] & a[j])

Quick Sort

- in-place algoritme: geen extra geheugen nodig (i.t.t. merge sort)
  kost vergelijkingen gem $\sim 1{,}39\, n \log_2(n)$
- merge sort: eerst recursie (triviaal) dan mergen (echte werk)
  quick sort: eerst partitie (echte werk) dan recursie (triviaal)

partitionering:
= kies een partitioneringselement dat de rij in 2 stukken deelt, vaag a[j] → pivot
→ 1 deel met alle elementen kleiner dan het pivot (links)
1 " " " " " groter of gelijk aan het pivot (rechts)
⇒ $a[lo] \dots a[j-1] \leq a[j]$
$a[j+1] \dots a[hi] \geq a[j]$

deze 2 rijen moeten nu onderling verder gesorteerd w
dan we deze bovenste strategie te herhalen, recursief verder sorteren
de pivot blijft ALTIJD op dezelfde plaats staan
[partitionering in echte werk]

```
public class QuickSort {
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);   // op voorhand shuffle zodat elk element evenveel
        sort(a, 0, a.length-1);   kans heeft om pivot te worden
    }
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;   → 1 (of 0) element over, uit recursie stappen
        int j = partition(a, lo, hi);   = partitionering, a[j] = pivotelement
        sort(a, lo, j-1);   recursief sort oproepen // sort linker deel: a[lo .. j-1]
        sort(a, j+1, hi);   // sort rechter deel: a[j+1 .. hi]
    }
}
```

invarianten:
- a[lo .. j-1] is nooit groter dan a[j]
- a[j+1 .. hi] is nooit kleiner dan a[j]
- a[j] is in zijn finale plaats in de rij

→ 3 partitioneringsvarianten

1. Simpele partitionering
   kies een pivot element
   maak 2 lijsten: a[i] < a[j] → left, a[i] > a[j] → right, [a[i] == a[j] → 1 vd 2 kiezen] (a[j] = pivot)
   → concatenate [left]+[pivot]+[right]
   ⇒ n-1 vergelijkingen
   nadeel: extra geheugen nodig voor 2 nieuwe arrays } inefficient
   + extra operaties voor kopiëren & concatenaten (lin.) }

2. Partitionering van Hoare
   kies 1e element als pivot
   2 lussen: i naar rechts doen lopen
   j naar links " "
   alle elementen kleiner dan pivot op pos i
   blijven staan & i loopt verder totdat we
   op pos i een element groter hebben → stop
   alle elementen groter dan pivot op pos j blijven staan & j loopt verder tot kleiner → stop
   → elementen a[i] en a[j] van plaats wisselen en opnieuw verder doen
   → stop wnnr i en j "botsen" → laatste kleinere element wisselen met pivot
   ↳ toepassen op eindes welke actief deel vd array

i = 1e element rechts vd pivot
j = laatste element

2. Hoare

```
private static int partition(Comparable[] a, int lo, int hi){
    int i = lo , j = hi+1        // left and right scan indices
    Comparable v = a[lo]         → pivot, partitioning value - 1e element
    while (true) {
        while (less(a[++i], v)){
            if (i == hi) break;          - scan right
        while (less(v, a[--j])){
            if (j == lo) break; }        - scan left
        if (i >= j) break;               — check for scan complete
        exchange(a, i, j); }             - exchange
    exchange(a, lo, j);  → put pivot v in position v = a[j]
    return j; }  → j is position of pivot, with a[lo..j-1] ≤ a[j] ≤ a[j+1..hi].
```

efficient
weinig
vergelijkingen

n+1 vergelijkingen

3. Partitionering van Lomuto

kies laatste element als pivot
index j = 1e element dat we moeten bekijken
index i = laatste element gevonden kleiner dan pivot

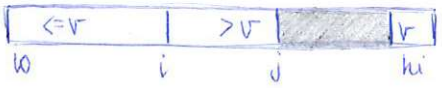
2 opties: 1) als a[j] > v: element mag blijven staan en j schuift 1 naar rechts

2) als a[j] ≤ v: element moet op positie i+1 komen te staan → a[j] excht. a[i+1] en i = i+1, en j = j+1.

herhalen totdat a[j] = v
⇒ pivot met a[i+1] wisselen

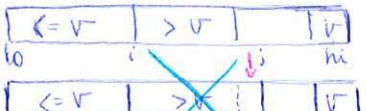
voor

tijdens

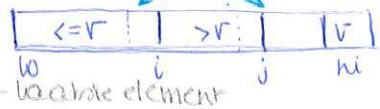


na

a[j] > v



a[j] < v




```
private static int partition (Comp[] a, int lo, int hi){
    int i = lo - 1;          → scan index
    Comparable v = a[hi];    → pivot, partitioning value - laatste element
    for (int j = lo; j <= hi-1; j++) {   scan alle elementen
        if (less(a[j], v)){              skip equal values
            i++;
            exchange(a, i, j); }
    }
    exchange(a, i+1, hi);    → pivot op pos i+1 zetten, v = a[i+1]
    return i+1; }            → positie v teruggeven, met a[lo..i] ≤ a[i+1] ≤ a[i+2..hi]
```

elegant

} 2 elementen w nooit meer dan 1 keer met elkaar vergeleken

n-1 vergelijkingen

Performantie quicksort

best case: pivot steeds mooi in het midden, gebalanceerd opsplitsen
stel dat de partitie n vergelijkingen uitvoert

$T(n) = 2T(\frac{n}{2}) + \text{partitioning}(n)$

$= 2T(n/2) + n \pm 1$

$= n \pm 1 + 2(\frac{n}{2} \pm 1 + 2T(n/4))$

$= n \pm 1 + n \pm 2 + 4T(n/4)$

$= n \pm 1 + n \pm 2 + n \pm 4 + 8T(n/8)$

$= \dots$

$\sim n \log_2(n)$ → wnl goed als merge sort, maar zonder extra geheugenschr.

+ → Hoare
− → Lomuto

tijd om n elementen te sorteren
= tijd partitioneren
+ tijd linkerij sorteren
+ tijd rechterij sorteren

recondeltelling
linker en rechterij gelijk
[wig $1\frac{n}{2}$ en $1\frac{n-1}{2}$]

worst case: 0 elementen in de ene subarray en n-1 in de andere
↳ pivot steeds kleinste (of grootste) element – wij al gewerkt
⇒ zelfde gedrag als selection sort
$T(n) = T(n-1) + T(0) + \text{partitionering}(n) = T(n-1) + \text{partitionering}(n) = \ldots$
$\sim n^2/2$

kans dat we in worst case zitten: kans $\frac{2}{k}$ dat we vanaf of vanachter de kleinste (sub)array pivot nemen

2 kansen op n
2 kansen op $n-1$
...
2 kansen op 2

$\frac{2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2}{n!} = \frac{2^{n-1}}{n!} = P(\text{worst case})$

→ $n!$ stijgt meer dan 2^{n-1}
vb $n=10$, $P = 0{,}014$

(laatste kan mag je niet tellen, want dan mag maar 1 elementje over)

average case: redenering dat split in in proporties van n
$\sim c\, n \log_2 n$

vb split @ 9/10 op elk niveau $T(n) = T(9n/10) + T(n/10) + \text{partitionering}(n)$
$\sim cn \log_2 n$ hier $c = \log_{10/9} 2$

n compares op level 1, $n/10 + 9n/10 = n$ compares op level 2
$n/100\, n + 9/100 n + 9/100 n + 81/100 n = n$ compares level 3 ...

Quicksort kan teruggekomen van 1 slechte split – 1 extra kost $\sim n$ (tegenover een team)
⇒ $n \log_2 n$ gedrag behouden ↳ $\sim n \log n + (\#\text{slechte stappen}) \cdot n$

"exacte" wiskundige benadering
aantal vgl obv # vgl nodig voor elke partitionering
aannames: elk element kwetst kans om pivot te zijn ($P = 1/n$)
x partitionering voor n elementen kost $n+1$ vgl (Hoare)
vgl'en nodig om de n elementen te sorteren:

$$C(n) = (n+1) + \frac{1}{n}[C(0) + C(n-1)] + \frac{1}{n}[C(1) + C(n-2)] + \ldots + \frac{1}{n}[C(n-1) + C(0)]$$

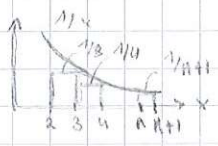
partitionering; $\frac{1}{n}$ kans dat we 0 el. i/d linkse en $n-1$ i/d rechtse partitie hebben; $\frac{1}{n}$ kans voor 1 i/d linkse en $n-2$ i/d rechtse; $\frac{1}{n}$ kans voor elke mogelijke uitkomst vld uitkomsten vld partitie

gem kost

$$\Leftrightarrow C(n) = (n+1) + \frac{2}{n}[C(0) + C(1) + C(2) + \ldots + C(n-1)]$$
$$\Leftrightarrow nC(n) = n(n+1) + 2[C(0) + C(1) + \ldots + C(n-1)]$$
voor $n-1$: $(n-1)C(n-1) = (n-1)n + 2[C(0) + C(1) + \ldots + C(n-2)]$
$$\Rightarrow nC(n) - (n-1)C(n-1) = n(n+1) - (n-1)n + 2C(n-1) = 2n + 2C(n-1)$$
$$\Leftrightarrow nC(n) = 2n + (n+1)C(n-1)$$
$$\Leftrightarrow \frac{C(n)}{n+1} = \frac{2}{n+1} + \frac{C(n-1)}{n}$$ → heel symm recurrente uitdrukking

$$\Rightarrow \text{expansie: } \frac{C(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots + \frac{2}{3}$$ [kost 0, 1, 2 elementen sorteren is $C(0) = C(1) = 0$, $C(2) = 1$]

$$\Leftrightarrow C(n) = 2(n+1)\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{3}\right)$$



$$\approx \int_2^{n+1} \frac{1}{x}\,dx = [\ln x]_2^{n+1}$$
$$\approx \int_2^{n} \frac{1}{x}\,dx + \frac{1}{n+1}$$ willen $\log n$ uitdr. t.o.v. $\ln(n)$
$$= [\ln x]_2^n + \frac{1}{n+1} = \ln(n) - \ln(2) + \frac{1}{n+1}$$

$$\Rightarrow C(n) = 2(n+1)\left(\ln n - \ln 2 + \frac{1}{n+1}\right)$$

tilde notatie
$$C(n) \sim 2n \ln(n)$$
$$\sim 1{,}39\, n \log_2 n$$

$\ln(n) = \log_e(n) = \log_e 2 \log_2 n$, $\log_e 2 = 0{,}61$

stel dat we Lomuto gebruiken ipv Hoare

$\rightarrow C(n) = (n-1) + \frac{2}{n}[C(0) + C(1) + \ldots + C(n-1)]$

$\Rightarrow n\,C(n) = n(n-1) + 2[C(0) + C(1) + \ldots + C(n-1)]$

$(n-1)\,C(n-1) = (n-1)(n-2) + 2[C(0) + C(1) + \ldots + C(n-2)]$

$\Rightarrow n\,C(n) - (n-1)\,C(n-1) = n(n-1) - (n-1)(n-2) + 2\,C(n-1)$

$= (n-1)(n-n+2) + 2\,C(n-1) = 2(n-1) + 2\,C(n-1)$

$\Rightarrow n\,C(n) = 2(n-1) + (n+1)\,C(n-1)$

$\Rightarrow \frac{C(n)}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{C(n-1)}{n}$

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

extra lagere orde term

$C(n) = 2(n-1)\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2}\right) \approx 2(n-1)\ln n \approx 2n\ln(n)$

$\approx \int_1^n \frac{1}{x}\,dx = \ln(n)$

uitkomst mee houden in recursieve expansie
wiskundig gerechtvaardigd dit ~

$\Rightarrow \sim 1{,}39\, n \log_2 n$

(bottom-up) aanpak 2 kans dat 2 elementen in de rij ooit met elkaar vergeleken zullen worden
optellen van deze kansen → probabilistisch verwacht # vgl

aanname Lomuto partitionering (nooit meer dan 1 keer 2 elementen met elkaar vergelijken

stel dat de elementen in de te sorteren rij reeds in volgorde gesorteerd zijn

$z_1, z_2, z_3, \ldots, z_n$ [verondersteld geen dubbele waarden]

wat is nu de kans dat z_i ooit met z_j zal worden vergeleken?

→ beschouw groep $\{z_i, \ldots, z_j\}$

- zolang pivot $v < z_i$ en $v > z_j$ zal de groep in één partitie samen blijven
- als $v > z_i$ en $v < z_j$ dan zal de groep in 2 gedeeld en tot verschillende partities behoren
  → z_i zal nooit met z_j vergeleken
- als $v = z_i$ of $v = z_j$ op een moment dat ze nog samen in een groep zitten
  → enkel dan z_i met z_j vergeleken

⇒ - als v buiten groep : blijven 1 partitie en gaan recursief verder
- als v in groep: z_i en z_j niet vergeleken ⇒ 0 vgl
- als $v = z_i$ of $v = z_j$ ⇒ 1 vgl, met de kans dat z_i of z_j de pivot is en zo dus vgl

elementen verder van elkaar → kleinere kans om ooit te vergeleken & vice versa

$P_{ij} = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$ → 2 kansen → # elementen in de groep

⇒ totale kost: $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$ ⇒ P_{ij} = # verwachte vgl voor z_i en z_j

$= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k}$

$= 2 \sum_{i=1}^{n-1} \left(\sum_{k=2}^{n-i+1} \frac{1}{k} \right)$

$\sim 2\,n\,\ln(n)$

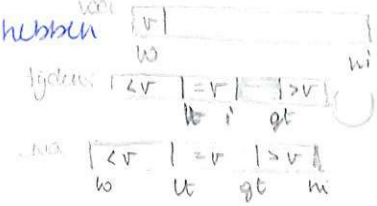
$\sim 1{,}39\, n \log_2 n$

) harm. som $H_N = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln(n) + 1$

→ quick sort duurt ongeveer 39% langer dan merge sort (wel geen extra geheugen nodig)

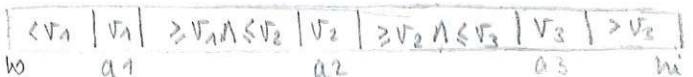
Optimalisering quick sort

* voor kleinere n (n < 10 meestal) insertion sort beter (vaginale orde termen spelen dan een rol)
* de mediaan van 3 willekeurige elementen uit de rij als pivot gebruiken
  → meer kans om in het midden te delen (best case)
* 3 way partitioning → wanneer veel elementen gelijkwaardige waarden hebben
  → less, equal, greater subarrays



* multi-pivot quicksort
  → meer werk per elementje, maar kleinere partitionering afwegingen maken

vb 3 pivots



comparison based sorteeralgoritmen kunnen niet beter dan $\sim n\log_2 n$
zoals we hiervoor hebben aangetoond
→ willen sorteren in lineaire tijd → bijkomende beperkingen!

- Bucket sort

de n elementen in k buckets verdelen $\sim n$
↳ elk element in zijn bucket steken gaat in cte tijd?
elke bucket is gesorteerd volgens een comparison based algoritme
* worst case: alle elementen in 1 bucket [slechte keuze buckets]
  ⇒ probleem niet verminderd — ↳ goede keuze bucket bel!
* best case: n/k elementen per bucket — gelijke verdeling input bel!
  └ komt overeen met average case waarbij we uitgaan van uniforme random input en dus een gelijke spreiding verwachten
  → comparison based algoritme $\sim cn\log_2 n$ toepassen op elke bucket
  $$\Rightarrow \sim k\left[c\,\frac{n}{k}\log_2(n/k)\right] = \sim cn\log_2(n/k)$$
  $$= \sim cn[\log_2(n) - \log_2(k)]$$
  • als k een cte is:
    ⇒ $\sim cn\log_2 n$ (geen verschil met quick / merge sort)
  • als k proportioneel is met n (k schaalt lineair met n)
    → n/k is een cte
    ⇒ $\sim c'n$ met $c' = c\log_2(n/k)$

keuze binnenste sorteeralgoritme
- weinig elementen per bucket → insertion sort
  (beginnen met sorteren elke keer een nieuw element i/d bucket is toegevoegd)
- veel elementen per bucket → quicksort of mergesort
- als opsplitsen voldoende is → geen sorteeralgoritme binnenin nodig

geheugengebruik: $\sim n$ extra ruimte
- array van k buckets
- elke bucket bevat een linked list van n/k elementen

- Key-indexed / Counting sort

relatief veel getallen sorteren maar met relatief weinig versch. waarden
= klein # keys → R = # versch. keys
→ tellen hoeveel elke key voorkomt ⇒ weten dat de eerste zoveel plaatsen die key gaan hebben, de volgende zoveel de volgende key, enz.

i	0	1	2	3	4	5	6	7
a[i]	a	d	c	a	b	e	a	e

r	a	b	c	d	e	-
count[r]	0	3	1	1	1	2
	0	3	4	5	6	8

0 keys kleiner dan a → a begint op a[0]
4 keys < c → c begint op a[4]

i	0	1	2	3	4	5	6	7
aux[i]	a	a	a	b	c	d	e	e

```
int N = a.length;
int[] count = new int[R+1];            → counting array ≈ teller
for (int i=0; i<N; i++)                tellen v/d freq. : lineair in N
    count[a[i]+1]++;                   [offset van 1]
for (int r=0; r<R; r++)                geaccumuleerde som bepalen
    count[r+1] += count[r];              ~ lineair in R
for (int i=0; i<N; i++)                hulparray om bij te houden hoeveel
    aux[count[a[i]]++] = a[i];         elementen al geplaatst zijn
for (int i=0; i<N; i++)                elementen terugplaatsen
    a[i] = aux[i];
```

? telalgoritme geen elementen met elkaar vergelijken → impliciete orde relatie
complexiteit: in termen van array accesses [waarde opvragen, wegschrijven of bewerking]
$\sim 8N + 3R + 1$ (om N elementen met keys integers tss 0 en R-1 te sorteren.
↳ kan variëren afh van hoe je telt, sowieso met $\sim N$ en $\sim R$

complexiteit: extra geheugen ruimte

$\sim N+R$ (counting array + auxiliary array)

* in-place sorteeralgoritme
  = sorteeralgoritme gaat niet extra data structuren zelf elementen verplaatsen
  → geen extra geheugen nodig
  vb selection sort, insertion sort...

  VS out-place sorteren (extra geheugen nodig)
  vb merge sort, counting sort...

* stabiel sorteeralgoritme
  = onderlinge volgorde van elementen met dezelfde key/waarde
  W NIET veranderd in finaal sorteren
  → onderlinge volgorde behouden
  ⇒ hiërarchisch sorteren mogelijk [vorige sortering blijft behouden]
  vb counting sort
  ↳ belangrijk voor het grote data vb strings karakter per karakter vergelijken in vergelijken traag gaat

- Least Significant Digit sort (radix sort) radix = basis (eenheden, tientallen en zo)
  ! alle getallen / strings moeten dezelfde lengte hebben
  begint de karakters van rechts naar links
  ↳ begin sorteren met LSD (laatste karakter ~ eenheden)
  dan volgende digit (nieuwe LSD, nu voorlaatste karakter ~ tientallen)
  herhalen tot LSD eerste karakter is
  gebruiken hiervoor telkens counting sort
  (aangezien counting sort stabiel is blijft de onderlinge volgorde via LSD's behouden

  $\sim (7N+3R)W$ aantal access
  $= \sim 7NW + 3RW$ (met W = woordlengte)
  $\sim N+R$ ruimte
  7 of 8 hangt af van initialisatie / hoe je telt
  bellll is dat het lineair verloopt

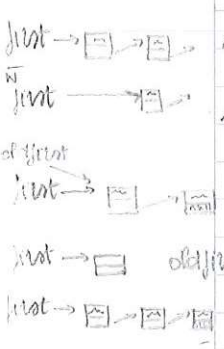
- Most Significant Digit sort
  ! kan strings van verschillende lengte wel sorteren ~ soort van bucket met counting sort
  begint karakters van links naar rechts
  ↳ karakter per karakter sorteren met counting sort beginnende met het eerste, dan recursief toepassen op het tweede, dan recursief op het derde enz. uiteindelijk dan terug alles samenvoegen
  ↳ in groepjes ingedeeld op het 1e karakter
  en elk van deze groepjes nu counting sort oproepen
  en dit per groepje blijven herhalen
  → groepjes onafhankelijk (recursief) van elkaar sorteren
  globaal gezien $\sim N \log_R N$ karakters onderzocht → MSD bepaalt met zoveel karakters om de keys te sorteren
  # onderzochte karakters afh van keys
  # strings
  # verschillende waarden per string
  # karakters in het gebruikte alfabet
  ↳ in elke counting sort elk karakter scheiden
  teller een niveau R naar beneden
  ! kan subliniaire zijn
  ↳ door recursie wel counting sort vaak opgeroepen
  voor klein # digits/strings kan deze voor een grote overhead zorgen
  ⇒ voor klein # strings insertion sort gebruiken
  omdat voor een beperkt karakters, maar wel nog steeds stapsgewijs

- 3-way string quicksort
  - werkt op strings
  - 3 partities: kleiner dan, groter dan of gelijk aan de pivot (niet 2 pivots)
  - vergelijken van strings zoals beschreven in MSD (1e karakter eerst)
  - maar dan quicksort uitvoeren groepen ipv counting sort
  - waardoor de R afhankelijkheid wegvalt
  - ⇒ ~ $1,39 N \log_2 N$ karakter compares (i.t.t. ~$1,39 N \log_2 N$ STRING compares standard quicksort)
  - strings op basis van 1e karakter opdelen in/d 3 partities
    - voor >, < pivot: recursief hetzelfde karakter gebruiken om de groep verder te sorteren
    - voor = pivot: recursief het volgende karakter gebruiken om de groep verder te sorteren
  - vermijd het opnieuw vergelijken van het begin vd strings
  - gebruikt niet evenveel compares
  - substantieel winst als de strings lang zijn

4.3 X 4.4

Stacks x Queues
- fundamentele datatypes: iets van objecten met basisoperaties: insert, remove, isEmpty
  - verzameling van elementen

- Stacks
  - element dat we verwijderen is het meest recent toegevoegde element ~ stapel
  - → push (element toevoegen) en pop (element verwijderen) gebruiken
    - nieuwe achteraan (of vooraan afh van implementatie) de verzameling
    - void push(Object o), Object pop(), boolean isEmpty()
    - lui nooit iets dat voor iets anders opgeslagen is eerder afhalen dan dat laatst object
  - * implementatie met gelinkte lijsten
    - linked list = datastruct. met elementen die aan elkaar gelinkt zijn met verwijzingen/pointers - alle elementen naast elkaar → linked list
      - → klasse Node met 2 datavelden
        - item: object met effectieve data [type String, Integer, ...]
        - next: pointer naar het volgende object [van type Node]
      - → Stack = 1 variabele [Node] bijhouden die verwijst naar het allereerste element vd gelinkte lijst, zeg first
      - → pop()



```
String item = first.item;   // uit object first [klasse Node] w item teruggegeven
first = first.next;         en first w overschrijven dr pointer next
return item;                (2e element w 1e (top))
```

- → push(String item)


```
Node oldfirst = first;
first = new Node();
first.item = item;          // first verwijst naar het nieuwe element
first.next = oldfirst;      // pointer toevoegen om van nieuwe 1e el nr 2e te gaa
```

- performantie:
  - pop cte tijd
  - push cte tijd
  - } N elementen pushen/poppen ~N tijd
  - stack met N items: ~N opslagen

* implementatie met arrays

reserveren een geheugenblok met array s[] voor de stack.
houden de plaats bij, zeg N, waar volgend vrij element is
→ push(item): s[N] = item; item toevoegen op plaats s[N]
pop(): element op plaats s[N-1] verwijderen

↳ voor intrieel int N = 0;
push(item) { s[N++] = item; }
pop() { return s[--N]; }

→ gemakkelijker want moeten nu geen pointer bijhouden, maar wel een geheugenblok reserveren

▽ resizen als array vol is (array heeft maar bepaalde capaciteit)
→ nieuwe array maken groter dan de vorige
moeten alle elementen vd oude array in de nieuwe kopiëren (KOST!)
we willen geen te grote (veel geheugen), maar ook geen te kleine (te vaak resizen) nieuwe array

optie 1 push op een volle array → vergroot lengte met 1
pop " " " " → verklein " " 1
[beginnen met stack grootte 0]
→ kost om een stack met N items te hebben
$N + 2(1+2+3+\dots+N-1) \sim N^2$ array accessen
kost voor elke nieuwe push op de stack
kost om tellers de array te resizen van k naar k+1 elementen $\sum_{k=1}^{N-1} k = \frac{(N-1)N}{2}$
→ kost kopiëren zijn tellers 2 array accessen [copy & paste]

optie 2 push op een volle array → verdubbel de lengte vd array
→ kost om de eerste N items op de stack te hebben
$N + 2(1+2+4+8+16+\dots+N/2) \sim 3N$ array accessen
kost voor elke push
kost om tellers de lengte te verdubbelen $\to N$
we maken minder vaak resizen en hebben langer een vaste kost per element

array doen krimpen om geheugen te besparen
we moeten oppassen dat de rij geen accordeon wordt door het veelvuldige pushen en poppen in de buurt van de volle array
⇒ lengte gehalveerd wanneer de array $\frac{1}{4}$ vol zit

Linked list vs dynamische array implementatie
linked list – worst case duurt elke operatie cte tijd
– gebruikt extra tijd en ruimte dit de pointers
dyn array – elke operatie duurt cte gemiddelde tijd
– minder geheugen verspilling

amortiseren = afbetalen door regelmatige aflossing

- Queues
element dat we verwijderen is het minst recent toegevoegde element, ~wachtrij
* implementatie met Linked lists
We hebben nu zowel een pointer naar het 1e als naar het laatste element
→ dequeue()
{ String item = first.item;
first = first.next;
return item; }
first = 1e object toegevoegd → head

→ enqueue(String item)
{ Node oldlast = last;
Node last = new Node();
last.item = item;
last.next = null;
oldlast.next = last; }
last = laatste object toegevoegd → tail

↳ allebei cte tijd
→ queue van N ~N tijd
~N geheugen

* implementatie met array
  array q[] om items in op te slaan
  → enqueue (item) : q[tail] = item;
    dequeue () : element op q[head] verwijderen
    → in head en tail steeds updaten modulo de capaciteit
      ook hier dynamic resizen

round robin system
als je elementen toevoegd aid tail en je komt op het einde vd array kun je je tail laten rondlopen dat je terug op pos 0 zit - doorlopen

- Hash tabellen (prim. database)
  elementen opslaan die we nu identificeren met een bep. sleutel
  → datarecords adv key (unieke identifier) plaatsen i/e array
    kunnen value (alle secundaire data) hebben
    in Obr die key ook opzoeken of verwijderen
  stel key is een integer
  → array met mapping sleutel 1 op 1 : value met key k op a[k]
  realiteit echter veel complexer (complexen sleutel, max(k) > array length, ...)
  ⇒ hash functie : manier waarop we de sleutels op posities vd tabel mappen
    - gecodeerde sleutel op juiste plaats
    ! surjectie : meer verschillende # sleutels dan plaatsen vd array
    → dealen met collisions (zie later)

hash functie
we mappen sleutel k op een integer ∈ [0, M-1] (met M = table.length)
eenvoudig
uniforme verdeling sleutels } ⇒ modulair hashen
zelfde sleutel → zelfde hash (simpelste hash functie)

modulair hashen
hash value = key % M
pos. int ↳ priemgetal (leidt tot minder collisions - minder hashes met dezelfde waarde)
stel M niet priem, dan als k = base 10 en M = 10^k → veel minder variatie in uitkomsten
willen juist veel variatie (minder collisions) → willen alle digits sleutel betrokken bij berekening
basisvoorstelling keys uniform verdeeld → hashes liefst ook zo uniform mogelijk verdeeld

collisions
tenzij M extreem groot is, zullen er altijd wel 2 sleutels op dezelfde index gehasht w
2 manieren om daarmee om te gaan
1) Seperate chaining
hash tabel interpreteren als een array v gelinkte lijsten
→ hash: mapt key op integer i tss 0 en M-1 met M = length array
insert: element vooraan in i^{de} linked list zetten (tenzij die er al in ~~die er nog niet staat~~ staat, mag blijven staan, wel value overwrite
search: i^{de} gelinkte lijst aflopen totdat element gevonden
willen de gem length vd linked list kort houden, want anders unhashen (hash tabel uitbreiden) en dat is veel een kost
we verwachten dat elke list $\alpha = N/M$ (load factor) elementen heeft
met N = # elementen , M = # plaatsen in hash tabel

uit observaties valt af te leiden dat de mate waarin de bezetting afwijkt van α binomiaal verdeeld is

⇒ kans dat k sleutels op dezelfde plaats gemapt $\tilde{w}$

$$\binom{N}{k}\left(\frac{1}{M}\right)^k\left(\frac{M-1}{M}\right)^{N-k}$$

met $\binom{N}{k} = \frac{N!}{k!(N-k)!}$

- $\binom{N}{k}$: permutatie zonder opties ("combinatie")
- $\left(\frac{1}{M}\right)^k$: kans dat k keys in 1 hokje
- $\left(\frac{M-1}{M}\right)^{N-k}$: kans N-k andere keys in andere M-1 hokjes

↳ zien dat de kansen snel afnemen om k keys op eenzelfde plaats te mappen als k afwijkt vd verwachte $k = \alpha$

⇒ # keys in elke linked list is ongeveer N/M [in de veronderstelling dat alle keys uniform mappen]

⇒ verwachte kost om elementen te zoeken $\sim N/M = cte$

want heel linked list aflopen

als M te groot → veel lege hokjes
als M te klein → ketens te lang
} kiezen typisch $M \sim N/5$ of $M \sim N/4$ ⇒ cte N/M ⇒ zoeken in cte tijd

verwachte kost om een element toe te voegen

* key is uniek aan element
  al een element met die key in vd hashtabel → oude element overschrijven met nieuwe el.
  bij toevoegen eerst checken of die key er al in zit of nog niet ⇒ $\sim N/M$
* key is niet uniek aan element, maar value (secundaire data) wel
  → element gewoon toevoegen vooraan aan de linked list ⇒ cte tijd

verwachte kost om elementen te verwijderen
worst case heel de rij aflopen om het te verwijderen element te lokaliseren
⇒ $\sim N/M$

resizing → rehashing: nieuwe hash tabel met meer hash waardes vereist enorm om de nieuwe hash vd elementen uit de oude hash tabel te berekenen en te plaatsen → KOSTELIJK
↳ doen we enkel als α boven bepaalde waarden
goal: $\alpha = N/M = cte$

! keys altijd bijhouden, hash is niet voldoende om specifiek element terug te vinden

2. Linear probing

lineair proberen = opeenvolgend zoeken waar de volgende lege plaats vd array is
N keys in tabel met grootte $M > N$ opslaan
(als het gewenste slot bezet is, neem je gewoon het eerstvolgende vrije (lineair aflopen)
↳ ! passen nooit meer elementen in je hashtabel dan er plaats is
zoeken vd tabel is nu complexer (weten op voorhand niet waar het element gaat zitten)
→ bereken hash
als key == search key → element gevonden
als hash een lege positie geeft → element niet in tabel
als key != search key → volgend slot proberen
herhalen totdat 1 vd eerste 2 opties voldaan is
} kan mogelijks veel tijd vragen

nadeel: neiging om clusters te vormen
hoe langer de cluster, hoe meer kans dat je element achteraan komt
→ nog langere cluster
+ mogelijkheid dat 2 clusters $\tilde{w}$ samengevoegd tot een supercluster
↳ kost zal groter $\tilde{w}$ (zoeken + toevoegen zal mogelijks langer duren)
⇒ willen clustervorming vermijden, bvb door quadratic probing

voordeel: zal altijd plaats vinden zolang tabel niet vol
+ geen gelinkte lijst nodig

resizing → weer alle keys rehashen
wanneer en hoe resizen?
gedrag linear probing: $\alpha = N/M < 1$ (door initiële aanname)
search hit (zoeken element in hash tabel, # pogingen nodig om te vinden)
$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$

zoeken mislukt (# pogingen om te bevestigen dat een element niet in de hash tabel zit = # pogingen voor inserts)

$\approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$

voor de keuze van $\alpha = 1/2$ wordt dit : zoeken lukt = 3/2
zoeken mislukt = 5/2

⇒ goal voor zoeken : $\alpha \ll 1/2$ ⇒ away verdubbelen of halveren indien nodig

element verwijderen uit hashtabel
maken "tombstone" achterlaten op plaats v/h verwijderde element (vb X)
deze markering zorgt ervoor dat we bij het zoeken naar een element weten dat hier iets verwijderd is en dus niet denken dat een bepaald element niet in de tabel zit wnnr het zich achter de tombstone bevindt.

Separate chaining	vs	Linear probing
- delete makkelijker te implementeren		- delete vraagt aandacht
- clustering minder gevoelig voor een slechte hashfunctie		- clustering kan een probleem vormen
- performantie redelijk goed		- minder verloren ruimte

2.4, 3.2, 3.3

- Priority queues
  = datastructuur (queue) waar we elementen op zetten, maar zo dat het grootste element vooraan staat (rest maakt niet uit ≠ gesort. zij)
  - bvat de basisoperaties insert(key), max(), delmax(), isEmpty()
    2 operaties die het eerste weg doen
  * als zij gesorteerd - makkelijk max bepalen
    delmax = laatste element afhalen ~1
    max = " " opvragen ~1
    insert : hele lijst aflopen om te weten waar in te voegen ~N
  * als zij ongesorteerd - makkelijk toevoegen
    insert : gewoon vanachter toevoegen ~1
    delmax, max : hele lijst aflopen ~N
  doel: willen insert en delmax aan elkaar gelijkstellen opdat zelfde complexiteit
  ⇒ gebruiken binaire boom als structuur

	insert	delmax / max		
ongesorteerde rij	1	N	N	→ vnl toevoegen
gesorteerde rij	N	1	1	→ vnl verwijderen
binaire heap	$\log_2 N$	$\log_2 N$	1	→ evenwicht

- Binaire boom
  → complete binaire boom : e/h element (buiten de bladeren) telkens 2 verbindingen
  ↳ diepte v/d CBT : N knopen → $1 + \lfloor \log_2 N \rfloor$ niveaus / diepte / hoogte
  # operaties beperkt dr diepte ($\log_2 N$)
  ↳ als nu elke knoop van die boom een bepaalde waarde heeft, en key, waarvoor geldt dat de key v/d ouder nooit kleiner is dan die van zijn kinderen, dan spreken we over een (max) heap
  hier meer bepaald een binaire heap

binaire boom = weg of knoop die linksen rechts met binaire bomen verbonden zijn

een binaire heap kan weergegeven worden als een boom, maar ook als een array
elke knoop is genummerd vanaf en vb no met de wortel niet zijnde 1
in het laatste staat N
deze nummers komen nu overeen met de positie van de keys in de array
→ * grootste key @ a[1]
* ouder van knoop k is op ⌊k/2⌋
* kinderen van knoop k zijn op 2k en 2k+1
kunnen geen uitspraken doen over de positie vd kleinste key
buiten dat het zich ergens bevind van ⌈N/2⌉+1 t.e.m. N → ouder van N is ⌊N/2⌋ en kleinste element heeft geen kinderen
de binaire heap kan nu gebruikt w als voorstelling voor een priority queue
→ max() { return a[1]; }
insert en delmax() maken gebruik vd functies swim() en sink()

- swim (↑)
stel dat je een geldige heap structuur hebt, maar
één knoop plots een waarde groter dan die van zijn ouder heeft
⇒ moeten ouder en kind van plaats wisselen
- moeten niet naar beneden kijken of de heap nog in orde is, want oudkind = ouder > oudouder = kind
- moeten wel checken of de nieuwe ouder nu niet groter is dan zijn ouder
→ als dit we zo is, moeten we iteratief dit proces herhalen totdat de ouder terug groter is of totdat we de root hebben bereikt.

```
private void swim (int k)                    → knoop die jouw zit (nummer)
{ while (k>1 && less(k/2, k))                  mogelijks
  { exch(k, k/2);
    k = k/2; }
}
```

k/2 is de ouder van knoop k

worstcase zit de wortel helemaal beneden ⇒ ~log₂N vgl'en
best case zit alles juist → ~1

↳ insert

```
public void insert (Key x)
{ pq[++N] = x;                  // knoop vanonder toevoegen (volgende vrije pos.)
  swim [N]; }                   // knoop en naar boven laten zwemmen
```

worstcase log₂N vgl'en

→ heap opbouwen
optie 1 elk element (vld N elementen) om de beurt in de heap insteken
heap w groter en groter
→ # vgl: (worstcase) $\log_2 1 + \log_2 2 + \log_2 3 + \dots + \log_2(N-1) + \log_2 N$
$= \log_2(N!)$
$\approx N \log_2(N)$ ← Stirling
⇒ ~N log₂(N) tijd (vergelijkingen)

- sink (↓)
stel dat één knoop plots kleiner is geworden dan minstens 1 van zijn kinderen
⇒ moeten ouder en grootste kind met elkaar wisselen
moeten dit blijven herhalen totdat de heap terug hersteld is

```
private void sink (int k)
{ while (2*k <= N)                          // blijven herhalen zolang knoop
  { int j = 2*k;                              nog kinderen heeft of tot break
    if (j<N && less(j, j+1)) j++;           // grootste vld 2 kinderen w j
    if (!less(k,j)) break;                  // vgl'en met grootste vld 2 kinderen
    exch(k,j);                              // kind w nu te checken ouder
    k = j; } }
```

2k 2k+1

```
private void sink (int k) {
    while (2*k <= N) {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (! less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

2 vergelijkingen per loop (kinderen met elkaar + ouder met oudere)
+ $\log_2 N$ niveaus
⇒ max $2 \log_2 N$ vergelijkingen

→ heap opbouwen

optie 2 elk element zit al in de heap/array
heap geldig maken door van onder naar boven (beginnende bij knoop $\lfloor N/2 \rfloor$, laatste ouder) zinken toe te passen (tot knoop 1) en bij elke stap steeds erover te zorgen dat alles terug in een geldige heapstruct staat
→ heap bottom-up geconstrueerd
worst case & conservatieve schatting:
zinken toepassen op $N/2$ knopen die elk max $\log_2 N$ uitv. kunnen zinken
$\sim N/2 \; 2\log_2 N \sim N \log_2 N$ vgl'en
echter overschatting want laatste uitv. kan max 1 uitv. zinken
⇒ meer exact: $\sim 2N$ vgl'en

↳ delMax
1e element is het max → verwijderen uit heap
en laatste element nu helemaal vanboven zetten en laten zinken

```
public Key delMax() {
    Key max = pq[1];
    exch(1, N--);          // N met 1 verminderen opdat we eigenlijk
    sink(1)                //   nog maar N-1 knopen hebben
    pq[N+1] = null;        // waarde vorige laatste knoop (die nu de
    return max; }          //   gezonken toptie) op null zetten
```

worst case $2 \log_2 N$ vgl'en

→ heapsort
n elementen die we willen sorteren in een ~~priority queue~~ array zetten
→ in gedalige heapstruct (opslaat pq)
- telkens grootste element verwijderen en achteraan in un array zetten
- heap 1 korter maken (vrijgerangt)
↳ n keer herhalen ⇒ gesorteerde array als resultaat

nadelige operaties: heapifying + (1 voor 1 delMax toepassen) N
$\sim 2N \log_2 N + \sim 2N \log_2 N$
⇒ $\sim 4N \log_2 N$

- Binaire zoekbomen (BST)
kan heel onregelmatige structuur hebben (qua bp diepte, telkens 2 vertakking maar mag ook leeg zijn)
enige voorwaarde: elke knoop i/d linkersubstab kleiner dan de
↳ symmetrische knoop erboven
oide elke knoop i/d rechtersubboom groter dan de knoop erboven
→ makkelijk om te doorzoeken - niet heel boom doorzoeken, maar telkens gewoon in juiste subtak blijven kiezen
gebruikt klasse Node met volgende var:

```
private class Node
{ private Key key;
  private Value value;
  private Node left, right;
  public Node (Key key, Value value) {
      this.key = key;
      this.value = value; }
```

- search : geeft value terug van gegeven key of null als de key er niet is
  voor hit: # vgl'en = diepte v/d knoop
  voor mis: # vgl'en = diepte van waar de knoop zou zijn
- insert : zoek de key in de boom
  - als de key v/d boom zit → value vervangen als er geen dubbels zijn toegelaten
  - als de key niet v/d boom zit → nieuwe knoop toevoegen op null link
    [blad onderaan in juiste subtak]

↳ veel verschillende BST's zijn mogelijk voor dezelfde set keys
vgl'en ≈ diepte v/d knoop (onvoorspelbaar)

gemiddelde gedrag v/e zoekboom als we willekeurig elementen toevoegen?
→ diepte ~ $1{,}39 \log_2 N$ → # vgl'en voor search hits ‖ realistisch echter onwaarschijnlijk diepte die willek. toevoegen

analyse analoog aan quicksort (pivot) met 2 partities (rechter knoop met 2 takken)

$$C(n) = 2 + \frac{1}{n+1}\left[C(0) + C(1) + C(2) + \ldots + C(n-1)\right]$$

2 vergelijkingen telkens
- gelijk aan knoop
- groter/kleiner dan knoop

bij afdalen kan de verzameling waarin we afdalen 0 elementen hebben, 1 element, 2 elementen, ..., n-1 elementen

elk van die scenario's hebben een gelijke kans om voor te komen

n+1 scenario's: element gevonden, tak met 0 elementen, 1 element, ..., n-1 elementen

$$\Rightarrow (n+1)C(n) = 2(n+1) + \left[C(0)+C(1)+\ldots+C(n-1)\right]$$
$$n\,C(n-1) = 2n + \left[C(0)+C(1)+\ldots+C(n-2)\right]$$

→ aftrekken: $(n+1)C(n) - n\,C(n-1) = 2 + C(n-1)$

$$\Rightarrow (n+1)C(n) = 2 + (n-1)C(n-1)$$
$$\Rightarrow C(n) = \frac{2}{n+1} + \frac{n-1}{n+1}C(n-1)$$

→

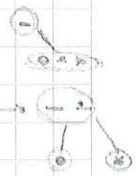
- delete :
  - simpel: maak de knoop leeg, maar laat hem staan gemarkeerd met een tombstone
  - delete min: ga zoveel mogelijk naar links tot je een null link links tegen komt
    vervang nu deze knoop door zijn rechterlink (en update de tellingen)
  - Hibbard deletion: zoek voor de knoop met key k die je wilt verwijderen
    * als k geen kinderen heeft: knoop gewoon verwijderen en door null link vervangen
    * als k 1 kind heeft: knoop verwijderen en link vervangen door de link naar het kind van k
    * als k 2 kinderen heeft: zoek de opvolger van k → minimum in rechter subboom verwijder dat minimum en vul de waarde van deze knoop in op de plaats van k

gemiddeld $\sim\sqrt{N}$

- order operaties:
  findMin → key uiterst links
  findMax → key uiterst rechts
  floor (= grootste key ≤ k) → als k == root: floor is root
  k < root: floor moet in de linkerboom zitten
  k > root: floor kan in de rechtersubboom zitten of de root zijn
  ceil (= kleinste key ≥ k) → gelijkaardig aan hierboven maar dan omgekeerd

L problemen BST's
- slechte worst case performantie
- idealiter: gebalanceerde boom → theoretische pogingen hiervoor: 2-3-boom & rood-zwart-boom

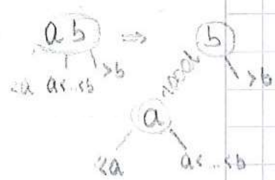
- **2-3 bomen** ander type search tree
    1 of 2 keys per knoop toegestaan
    → 2-knoop : 1 key & 2 kinderen
    3-knoop : 2 keys & 3 kinderen (kleiner dan L, groter dan R, tss L en R)
    → PERFECT GEBALANCEERD : elk pad van root tot blad heeft dezelfde lengte
    ↳ mogelijk 2 testen doen om te weten in welke tak we moeten afdalen
    → search 2 testen / vgl'en om niv dalen te kiezen, maar diepte niv wel vast
    • insert : element K toevoegen vanonder i/d boom (bladeren)
    ↳ opbouw ↳ in welke subtak bepaald dr vbn o te lezen
    * als de eindknoop waarin we terecht komen een 2knoop is
    → w̄ 3knoop (K gewoon toevoegen)
    * als eindknoop een 3knoop, met een 2knoop als ouder
    → tijdelijk 4knoop maken met 2knoop als ouder
    → middelste waarde (v/d 3) v/d 4knoop in de 2knoop ouder toevoegen en de 4→3knoop opsplitsen in 2 2knopen ↘ ouder w̄ 3knoop



STEEDS ELEMENTEN VANONDER TOEVOEGEN EN NAAR BOVEN TREKKEN

    * als eindknoop 3knoop met 3knoop als ouder
    → zelfde idee als hierboven, maar dit dan blijven herhalen totdat de ouderknoop een 3knoop is (en geen 4knoop meer)
    * als eindknoop 3knoop en we botsen op de root die ook een 3knoop is
    → niveau bijmaken
    boom laten groeien naar boven toe
    ↳ elke transformatie behoudt de perfecte balans en symmetrische orde
    ⇒ elk pad van root tot blad heeft dezelfde lengte
    → overal dezelfde diepte
    ↳ worst case : allemaal 2knopen → $\log_2 N$ diep (bovengrens)
    best case : allemaal 3knopen → $\log_3 N \approx 0{,}631 \log_2 N$ diep (ondergrens)
    → we garanderen dat search & insert algoritmes altijd logaritmisch gaan
    ⇒ search, insert, delete & search hit $\sim c \log_2 N$ met $0{,}631 \leq c \leq 1$

- **Rood-zwart bomen** (RB trees)
    = 2-3 bomen vertalen in binaire bomen (BST)
    → verbindingen in binaire boom kleur geven



    * rood : ⓐ en ⓑ vormen samen een 3knoop → n̄ links leunende verbinding
    * zwart : gewone verbinding
    ↳ een veld i/d implementatie v/d knoop v/h kind houdt kleur verbinding bij
    "regels"
    - geen knoop mag 2 rode verbindingen hebben (of meer) ‖ enkel 2- of 3knopen in 2-3boom
    - elk pad v/d root naar een blad heeft hetzelfde # zwarte links (black-balance) ‖ 2-3 bomen overal vaste diepte
    - rode verbindingen altijd n̄ links
    ↳ variaties op mogelijk (zie rotaties)
    → we alle operaties die we op 2-3 bomen toepassen (bvb uitgebreide insert functie) ook op RB bomen toepassen (roteer links indien nodig)
    ↳ als 2 rode in 1 rij (2 opeenvolgende linker (of rechter) subtakken) dan mbv rotatie het middelste element "naar boven brengen" dmv rotaties en dan kleur terug n̄ zwart veranderen → flip! (indien middelste zwarte verbinding met ouder → w̄ rood)
    ! naar boven halen → ook zwart en rood manipuleren

max diepte RB boom : $2 \log_2 N$ (rode verbindingen erbij)
gem $\sim \log_2 N$

Greedy algorithms

5.5

= gulzige algoritmen → zetten telkens een stap die op dat moment het meest optimaal lijkt
(hopend dat je zo een goede uitkomst hebt)
→ probleem nooit globaal bekijken, maar opdelen in stapjes en die lokaal telkens bekijken
⇒ geen garantie op globaal het beste algoritme

vb coin exchange probleem: wisselgeld teruggeven in zo weinig mogelijk muntjes/briefjes
greedy algoritme: bij elke stap steeds het grootst mogelijke briefje/muntje teruggeven (zonder overshoot)
↳ werkt voor de meeste muntstelsels als meest optimale oplossing
echter voor bvb fictieve " $\{1, 7, 10\}$ werkt dit niet meest optimaal
↳ zal 15 teruggeven als $10+5\cdot1$ ipv meest optimale $2\cdot7+1$

Maffati probleem: 3 ingeschreven, niet-overlappende cirkels in een driehoek plaatsen opdat zo dat de cirkels zo'n groot mogelijk oppervlak bezetten
→ greedy algoritme geeft hier de meest optimale opl 

2 toepassingen bestuderen

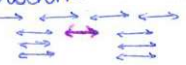
1 Activity Scheduling

zo veel mogelijk niet-overlappende activiteiten plannen in beperkte resource
constraint: begin- en einduur van elke activiteit
→ greedy algoritme opstellen, maar welk criterium gebruiken we om "de beste" activiteit lokaal te kiezen?

allemaal suboptimaal

Heuristiek 1: activiteit die het 1e begint als 1e plannen
2e activiteit ⇒ activiteit die na 1e het vroegst begint
enz.
! geen garantie op meeste # activiteiten i/h zwinkbesten
vb het kan zijn dat de 1e activiteit 10h duurt, waarin eigenlijk ook 4 andere activiteiten gepland konden w

Heuristiek 2: kortste (niet-overlappende) activiteit steeds kiezen
! geen garantie meeste # activiteiten
vb kortste activiteit v. 3 act. kan met de andere 2 overlappen waardoor je maar 1 act. kan plannen ipv 2 (tegenvoorbeeld)

Heuristiek 3: activiteit met minst # conflicten steeds kiezen
! geen garantie
vb  kiezen paarse activiteit wel maar kunnen dan maar 3 act. plannen ipv 4

Heuristiek 4: steeds de activiteit die als 1e eindigt en niet met de vorige overlapt, plannen
TB ⇒ OPTIMALE OPLOSSING (volgens # activiteiten)
MEEST
↳ misschien niet uniek maar wel opt.

Bewijs uit het ongerijmde
Stel $A = \{a_1, a_2, \dots, a_n\}$ de set van activiteiten gekozen volgens heuristiek 4.
[Deze activiteiten zijn gesorteerd volgens einduur (en dus ook volgens beginuur).]
Veronderstel nu dat er een andere set bestaat, bvb
$A' = \{a'_1, a'_2, \dots, a'_m\}$ met $m \geq n$
die meest optimaal is. ↳ optimaal → meeste # act
We vergelijken nu A' en A. willen bewijzen dat $m = n$
Veronderstel dat $a_1 \neq a'_1$,
dan eindigt a_1 voor (of gelijktijdig met) a'_1 [heuristiek 4]
dus dan kan a_1 NIET overlappen met a'_2.
Dan is $\{a_1, a'_2, a'_3, \dots, a'_m\}$ ook een geldige optimale opl.
Via inductie kunnen we zo alle a'_i's vervangen door a_i, $i = 1, \dots n$.
↳ kan a'_i in A' vervangen die a_i zonder de optimaliteit te beïnvloeden
$\Rightarrow \{a_1, a_2, \dots, a_n, a'_{n+1}, \dots, a'_m\}$ is dan ook een geldige optimale oplossing
Dit is echter in contradictie met heuristiek 4, want we zouden geen activiteiten meer mogen zijn die ~~overlappen~~ niet overlappen met a_n (constructie A), dus $a'_{n+1}, \dots, a'_m$ zijn tegenstrijdig
$\Rightarrow m = n$ ⇒ heuristiek 4 geeft een optimale opl. □

2 File compression

je strings, woordschap B in binaire data, in uw zo optimaal mogelijk # bits comprimeren, resp $C(B)$
daar decomprimeren kan je dan van $C(B)$ terug B verkrijgen
↳ compressieratio = $\frac{\text{bits in } C(B)}{\text{bits in } B}$ → willen dit zo klein mogelijk

! het ultieme compressiealgoritme bestaat echter NIET

IB → geen enkel algoritme kan elke bitstring comprimeren

Bewijs 1 dit contradictie

Stel dat er een compressiealgoritme bestaat dat elke bitstring kan comprimeren tot een kleiner # bits.
Dan kan een gegeven bitstring B0 gecomprimeerd $\bar{w}$ tot B1, maar dan kan B1 ook gecomprimeerd $\bar{w}$ tot B2 en dit kan herhaald blijven $\bar{w}$ totdat we uiteindelijk met een bitstring eindigen van lengte 1
→ Tegenstrijdig!
Alr we alle bitstrings kn compr. tot 1^{bit}, kn we ze niet meer decompr. □

we gaan hier uit van verlies compressie - dus zonder verlies van info

kn 1 bit niet laten decompr. tot zoveel bitstrings

Bewijs 2 dit te tellen hoeveel mogelijke gecomp. files je kan houden met een vast # bits

Stel dat er een algoritme bestaat dat alle mogelijke bitstrings ~~van~~ van 1000 bits kan comprimeren.
2^{1000} mogelijke bitstrings van 1000 bits (elke bit 2 verschillende waarden)
Het algoritme zal gecomprimeerde strings kn teruggeven van
1 bit lang → 2 zo'n strings, 2 bits lang → 4 strings, ..., 999 bits lang → 2^{999} strings
→ we zullen nu

$$\underbrace{1}_{\text{string v. 0 lang}} + 2 + 4 + \ldots + 2^{998} + 2^{999} = \sum_{i=0}^{999} 2^i = 2^{1000} - 1$$

(alle mogelijke strings voor te stellen met minder dan 1000 bits)

strings uit het algoritme kunnen volgen
Het ≤999 bits kn er namelijk maar $2^{1000}-1$ strings
gëncodeerd $\bar{w}$ ⟹ informatie verloren
⟹ gaat niet □

voorbeelden van algoritmen om bitstrings te comprimeren

- Run length encoding
  tellen hoeveel opeenvolgende nullen we hebben in bitstring i/h begin dan het # " enen, dan weer het # nullen en zo voort
  → in plaats van alle nullen en enen bij te houden houden we dus het #nullen, #enen, #nullen, #enen ... bij
  elk van die optellen houden we bij in x-bits (vb 4-bit telling voor max 15) (1111)
  ⇒ we kn zo bvb een string van 40 bits bijhouden in 16 bits
  hiermee kn we ook steeds onze oorspr. string terug opbouwen
  aangezien we onze getallen steeds moeten voorstellen als bits vr vaste lengte dan het zijn dat we soms 0 nullen of 0 enen moeten voorstellen → verlies van gegevens ⇒ niet optimaal (vb 17 nullen maar maar 4 bit reserveren of beginnen met enen ipv nullen)
  ↳ alternatieven
- variable length coding
  "codewoorden" kn een verschillende lengte hebben, maar dan moet je onderscheid kn maken tss de verschillende karakters om te weten wnnr een karakter in een string stopt/eindigt → delimiter introduceren
  vb spatie/gaap bij morse

vb morse of UTF-8 waarbij veel voorkomende karakters dan een kleiner # bits is voorgesteld

code = voorstelling kar.

- Huffman coding
  - meest optimale coding om bitstrings te comprimeren
  - = prefixvrije code → geen enkele code is een prefix / begin van een andere code
    - geen enkele code is de start van een andere code
    - ⇒ geen nood aan een delimiter
  - voor te stellen door een complete binaire boom waarbij elke vertakking overeenkomt met een 0 of 1 (links: 0, rechts: 1) en elk karakter correspondeert met een blad v/d boom — NIET met andere knopen die zich niet op het einde bevinden
  - beste code voor elk karakter opdat de tekst in zijn geheel zo kort mogelijk geëncodeerd is
- → Huffman coding
  - prefix vrije code construeren a.h.v. greedy algoritme
  - tel de frequentie van elk karakter in de te coderen sequentie
    - → 2 minst frequente karakters linken mbv een nieuwe knoop
    - nu dit telkens herhalen (minst frequente karakters eerst samen combineren) totdat alles gecombineerd is in 1 codeboom

vb 

1) 2 minima van pq afhalen
2) combineren onder een nieuwe knoop; freq. knoop = Σ freq. bladeren
3) 1 & 2 herhalen totdat alle knopen met de laagst mogelijke freq. gecombineerd zijn
4) 1 & 2 herhalen, maar nu ook de al eerder gemaakte vertakte knopen gebruiken v/d combinatie

⇒ eindresultaat = codeboom

implementatie vereist een priority queue om telkens de 2 karakters met de laagste freq te vinden ⇒ ~n log₂ n tijd nodig om code op te stellen voor n karakters

! BOOM moet COMPLEET zijn: als een knoop vertakkingen heeft, moeten het er 2 zijn, anders kan de prefixvrijheid niet gegarandeerd w

Om te bewijzen dat de Huffman codeboom optimaal is, zullen we eerst enkele lemma's beschouwen

Lemma 1 Een optimale codeboom is compleet

B altijd 2 vertakkingen als er vertakkingen uit een knoop zijn anders kan de boom collapsen tot een optimalere boom ⇒ compleet

Lemma 2 Er bestaat een optimale codeboom in dewelke dat de 2 minst frequente karakters broer en zus zijn op maximale diepte

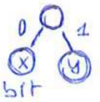
(niet uniek kan ook anders, maar dit is er zeker één)

B Uit lemma 1 volgt dat elke knoop op het onderste niveau een broer of zus heeft. Zeg nu dat x en y de minst frequente karakters zijn, dan moeten zij op maximale diepte v/d codeboom zitten. Want stel dat dit niet zo was, dan zou er een letter w zijn op maximale diepte die meer frequent voorkomt (veronderstelling x, y) en dat zou niet optimaal zijn.
Stel nu dat x en y geen siblings zouden zijn, dan kan dit gevormd w door y te wisselen met x's huidige sibling. Dit verandert de code, maar niet het # bits
⇒ kan altijd zo een boom maken. □

Stelling De Huffman codering zal leiden tot de meest optimale codeboom.

B. dmv inductie

basisstap: 2 karakters w volgens Huffman in deze codeboom voorgesteld



elk karakter w hier voorgesteld dmv 1 bit
⇒ optimaal

inductiestap: veronderstel nu dat we de optimale codeboom kunnen opstellen voor minder dan r karakters
we tonen nu aan dat de codeboom voor r karakters dan ook opt. is

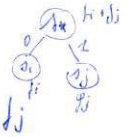
(zie volgende pagina)

optimale boom T_H^* voor $n-1$ har. uit breiden tot n karakters $\rightarrow T_H$

Stel dat T_H de Huffman codeboom is voor karakters $s_1, \dots, s_n$ met respectievelijk freq. $f_1, \dots, f_n$
Om T_H op te stellen beginnen we met de 2 symbolen met de minste freq te kiezen, zeg s_i en s_j
Deze 2 combineren we dan in een gezamenlijke knoop, zeg s^* met freq $f_i + f_j$
Nu zijn er nog $n-1$ symbolen over die we moeten combineren volgens Huffman.
Aangezien we veronderstellen dat Huffman een codeboom die optimaal is geeft voor alle karakters t.e.m. $n-1$, liggen we door Huffman toe te passen op de overige karakters de optimale boom T_H^*.
Dan is de totale lengte (= tot # bits) van resulterende bitstring
$$W(T_H) = W(T_H^*) + (f_i + f_j)$$
↑ elke lengte s_* uiteindelijk 1 langer, $f_i + f_j$ keer
geeft lengte bitstring $n-1$ symb, met s_*

optimale willekeurige boom T_{opt} voor n karakters comprimeren tot $n-1$ kar $\rightarrow T^*$

Stel nu dat er een andere ~~optimale~~ boom T_{opt} bestaat voor $(s_1, f_1), \dots, (s_n, f_n)$ die optimaal is
Uit lemma 1 en 2 volgt nu dat de minst freq knopen s_i en s_j op het diepste niv. als broer en zus voorkomen
Maak nu een nieuwe boom T^* door s_i en s_j te mergen in hun ouder $(s^*, f_i + f_j)$ → nieuw symb
T^* bevat nu $n-1$ symbolen
Dan is de totale lengte vlgd meest opt boom
$$W(T_{opt}) = W(T^*) + (f_i + f_j)$$
Uit de inductiehypothese (T_H^* is opt. voor $n-1$ kar.) vinden we dan dat
$$W(T_H^*) \leq W(T^*)$$
En aangezien $W(T_H) = W(T_H^*) + (f_i + f_j)$
vinden we dat $W(T_H) \leq W(T^*) + (f_i + f_j)$
$$\Rightarrow W(T_H) \leq W(T_{opt})$$
Dus $W(T_H) = W(T_{opt})$ (def optimale boom) en dus T_H is een optimale codeboom voor n karakters.
Uit de inductie volgt dus dat de Huffman boom optimaal is voor elk # karakters > 2. □



0, 1 van je bits bepalen niet je code, kan bvb vervangen in dit a,b een code is verschillend van een andere code als je voor je letters een andere lengte van codes krijgt

Minimum opspannende bomen

F.V.I.

grafen: knooppunten (vertices) met onderlinge verbindingen (bogen/edges), eventueel met een gewicht
↳ paden tss 2 knopen?
ring vld graaf? [Eulerraanschijving: alle bogen exact 1 keer gebruiken
Hamiltonraanschijving: alle knopen exact 1 keer gebruiken]
isomorfismen?
biconnectiviteit?

- Minimum opspannende boom
  = een ~~opspannende boom~~ subgraaf T voor graaf G die alle knopen verbindt ~~zonder een ring te maken~~ met minimale kost (som gewichten gebruikte bogen G (positief))
  - MST is niet uniek

voorstelling grafen i/e gegevensstructuur - nodig om operaties uit te voeren

* lineair array of linked list vld bogen
  → houdt de koppels vld verbonden knopen bij
  ⊖ alle bogen ~~zijn opschrijven vanuit 1 knoop~~ in/uit een knoop
  ⇒ lineair ~ e → geen goede struct.
  geheugen ~ e
* 2D-array: 2D V×V boolean array
  → verbinding tss i en j: x_{ij} = true (1)
  geen boog tss i en j: x_{ij} = false (0)
  ⊕ bogen tss 2 knopen → cte tijd ~ 1
  ⊕ alle bogen in/uit een knoop → lineair ~ v [i.v. ss ~ e als vld graaf] (# v en # e scalen vaak in zelfde
  ⊖ veel geheugen, ook als heel spaars
  ⊖ houden hele matrix bij ondanks symm (i,j) = (j,i)
  ⊖ updaten = updaten van heel de matrix
  geheugen ~ v²
* lijst of array die voor elke knoop v een geordende lijst heeft die bijhoudt naar welke andere knopen u een verbinding is
  ⊕ alle bogen in/uit een bep knoop → cte tijd
  ⊕ geheugen relatief goed
  ⊖ redundantie, maar nodig om voordelen hierboven goed uit te voeren
  ↳ kopiëren referenties naar obj die bogen voorstellen, niet naar bogen zelf
  geheugen ~ v + e

- MST algoritmes: allemaal greedy algoritmes die gebruik maken vld

cut property

↳ een snede = een partitie vld vertices (knopen) in 2 niet-lege verzamelingen
→ deelt knopen graaf op in 2 groepen
met de zogenoemde "crossing edges" die knopen vld 2 verschillende groepen met elkaar verbinden

gaan uit vle samenhangende graaf met duidelijke gewichten per boog

↳ cut property = van alle bogen die door een snede gaan (alle crossing edges), (eender welke) meer dingen met het minimaal gewicht deel uitmaken vld MST

Bewijs uit het ongerijmde
Stel e is de min-weight crossing edge
Stel dat e niet in de MST T zit, maar dat de verbinding gemaakt is door een andere crossing edge, zeg f.
Dan $w(f) \geq w(e) \Rightarrow w(T \text{ met } f) > w(T' \text{ met } e)$
waarin T' de opgespannen boom is verkregen door f te verwijderen en e toe te voegen, en T de MST met f is.
Dit is echter in tegenstrijd met de def van een MST.
⇒ e moet vld MST zitten.

□

→ greedy MST algoritme
vertrek vanuit alle bogen, waarvan er nog geen in de MST zitten (MST init. leeg)
maak een snede door bogen die nog niet in de MST zitten
→ voeg de doorgesneden boog met min. gewicht toe aan MST
herhaal totdat er V-1 bogen in MST zitten
↳ hoe kiezen we nu deze snedes? → vorige alg.

→ Algoritme van Prim
- start met 1 bep knoop, zeg vertex 0, in MST T
laat T gelijkig groeien
↳ gebruiken als snedes de snedes die alle bogen (v,w) met v ∈ MST, w ∉ MST ~~kruising volgen~~ doorsnijden
steeds de boog die aan 1 kant in en aan de andere kant uit T zit en min. is

↳ Lazy Prim: implementatie adv priority queue ($\sim \log_2 n$)
laten oude bogen erin zitten en bijven gewoon de nieuwe (v,w) toevoegen
- zetten bogen (v,w) in PQ en nagen zo heel gemakkelijk de boog met het minimale gewicht op
- ! echter: niet alle bogen in PQ blijven geldig om toe te voegen
want oudere bogen kunnen kruisingen veroorzaken
→ checken bij selecteren min of beide knopen vd boog niet al in T zitten
indien wel zo → PQ opnieuw organiseren zonder die boog ($\sim \log_2 n$)
en opnieuw min. selecteren
↳ dit kan voor overhead zorgen
maar kost om telkens weer vd. nieuwe PQ op te stellen te hoog

analyse
worstcase queue kan max E bogen bevatten ~ PQ waar veel kleiner dan E
→ boog deleten / toevoegen $\sim c \log_2 E$
ruimte nodig voor PQ $\sim E$
⇒ alle bogen overlopen: tijd $\sim E \log_2 E$
ruimte $\sim E$

Eager Prim ↳ beter Prim (overbodige bogen niet langer bijgehouden)
houden PQ van knopen bij die door een boog met T verbonden zijn
als prioriteit houden we het gewicht vd kortste boog die de knoop met T verbindt bij [goedkoopste verbinding]
min element v verwijderen en bijhorende knoop toevoegen aan T vb (v,w)
- PQ updaten: voeg nu alle knopen x toe aan PQ die verbonden zijn met de toegevoegde knoop v
als x ∈ T: negeer
als x ∉ PQ: voeg x toe aan PQ
als x ∈ PQ: update de ~~prioriteit~~ prioriteit van x als (v,x) korter dan de al bestaande verbinding

→ voordeel max V elementen in PQ, al wel iets meer tijd die updaten
analyse: ruimte $\sim V$
tijd $\sim E \log_2 V$
} iets beter, vooral voor dense grafen, voor sparse echter $E \sim V$

→ algoritme van Kruskal
bogen beschouwen in oplopende volgorde (vb alle bogen in PQ en PQ steeds updaten)
voeg nu telkens de volgende boog (globaal min boog) toe aan MST T
TENZIJ dit een kring kan veroorzaken
→ directe algoritme nodig

detectie/lusalgoritme gebaseerd op union find:
codeer tot welke substructuur elke vertex al behoort
i/h begin allemaal -1 want nog niets geselecteerd
elke keer dat je een boog toevoegt, moet je telkens eerst de substructuur vd knopen opvragen
als de knopen in verschillende bomen liggen → mogen met elkaar verbonden w dre 1 vd 2 knopen waarde te geven vd andere knoop en de waarde vd andere knoop te verlagen met 1 (→ -2)

analyse kruskal: in principe mogelijk elke edge te relecteren
→ tijd ~ E log E
i/h algemeen trager dan Prim
toepassing: hiërarchische clusterstructuur opbouwen

kortste pad algoritmen

[in gerichte grafen → bogen hebben naast een gewicht ook een richting]

we gaan ervan uit dat er een kortste pad bestaat van startknoop s tot elke andere knoop v
→ goal: dit pad vinden
- we zien dat er steeds een kortste padboom (SPT) opt bestaat
→ SPT voor te stellen a.h.w. 2 arrays: edgeTo[ ] en distTo[ ]

vertex-indexed: edgeTo[ ] = [null | 5→1 | 0→2 | 7→3 | 0→4 | 4→5 | 3→6 | 2→7]
distTo[ ] = [0 | 1,05 | 0,26 | 0,97 | 0,38 | 0,73 | 1,49 | 0,60]

edgeTo[v] = laatste boog op kortste pad van s naar v
distTo[v] = lengte v/h kortste pad van s naar v

- clue kortste pad algoritmen: edge/boog relaxatie
stel we hebben pad s → v
en we hebben een pad s → w
als de verbinding v→w een korter pad oplevert dan de huidige verbinding s→w
⇒ laatste boog die in s→w naar w wijst (edgeTo[w]) wegdoen
en kortste padboom updaten met v→w
not als e = v→w (relaxatie boog) korter pad naar w via v, update distTo[w] en edgeTo[w]

- Heel algemeen SPT algoritme
initialisatie: distTo[s] = 0 en distTo[w] = ∞ ∀ w ≠ s
herhaal: dan: relaxeer eender welke boog (→ kijken wat kortste afstand tot een knoop is, die boog toevoegen aan SPT, eventueel andere boog updaten die betere verbinding maken die nieuwe knoop)
herhaal tot dat voor elke w het kortste pad vanuit s is berekend

Bewijs Doorheen het algoritme is distTo[w] de lengte v/e simpel pad van s naar v en
edgeTo[w] de laatste boog in dat pad
Elke succesvolle relaxatie verlaagt distTo[w] voor een bep w.
distTo[w] kan hoogstuit een eindig # keer verminderen → stopt ooit → distTo[w] min ∀w □

↳ welke bogen we relaxeren hangt af v/h algoritme
↳ optimaliteitsvoorwaarden:
distTo[ ] zijn de kortstepad afstanden vanuit s
and * voor elke knoop v: distTo[v] de lengte is van een pad van s naar v
* voor elke boog e = (v→w): distTo[w] ≤ distTo[v] + e.weight()
(geen verdere relaxatie mogelijk)

- Dijkstra's algoritme
bezoek alle vertices in stijgende orde van afstand tot s (→ niet-SPT vertex met de laagste distTo[ ] value)
voeg de laagste vertex toe aan SPT en relaxeer alle bogen verbindende uit die vertex
herhaal voor alle vertices

Bewijs Dijkstra gudig
elke ~~vertex~~ boog $e = (v \to w)$ wordt precies 1 keer gerelaxeerd (wanneer v toegevoegd a/d SPT), dus $distTo[w] \leq distTo[v] + e.weight()$
Deze ongelijkheid blijft gudig totdat het algoritme stopt omdat
- $distTo[w]$ niet kan stijgen; we hebben enkel niet neg. gewichten
- ~~$distTo[v]$~~ niet kan veranderen; v wordt niet meer gerelaxeerd

Dus tot bij het eindigen is de kortste pad optimaliteitsvoorwaarde gegarandeerd □

analyse
lgchcomplexiteit $\sim E \log_2 V$
want steunt (net zoals Prim) op een PQ van V knopen
in elke boog kan mogelijk eens gerelaxeerd w in SPT → update PQ E keer

het is met Dijkstra ook mogelijk om het langste pad te bepalen
door alle gewichten negatief te maken en het kortste pad te zoeken
echter: Dijkstra werkt niet met neg. gewichten, want dan kan je door een lus blijven lopen eindeloos je gewicht in die lus blijft afnemen
→ alternatief algoritme dat wel met neg. gewichten werkt
vb Belman-Ford
≈ Dijkstra, maar houdt ook het # keer dat een knoop gerelecteerd w bij omdat je nooit in een loop terecht kan komen → extra geheugen

Als we in gewichtte acyclische graaf hebben (geen lussen mogelijk die toegelaten zicht bogen)
→ wel met neg. gewichten werken met Dijkstra

kritisch pad = kortste pad van begin tot einde
hier vertraging → overal vertraging

voorbeeld winkelhaken: kortste pad voor beste winkelhaken t.o.v. 2 munten
kortste pad alg. werken echter met optellen
en winkelhaken met vermenigvuldigen
→ logaritme vd gewichten nemen
want min som log ≈ min vermenigvuldiging

5.3

Substring search

→ klasse van algoritme die gegeven een zeer lange string (tekst) van lengte N een bepaald patroon van lengte M probeert te vinden met M <= N

substring = stukje vd tekst

3 + 1 mogelijke algoritmen

0) Brute force

elk karakter vh patroon vergelijken met een karakter in je tekst, dan vr. het volgende enz.

```
public static int search(String pat, String txt)
{ int M = pat.length();
  int N = txt.length();
  for (int i=0 ; i<=N-M ; i++) { int j;
      for (j=0 ; j<M ; j++)
          if (txt.charAt(i+j) != pat.charAt(j))
              break; }
      if (j==M) return i ; } → index vd tekst waar pat. start
  return N; → patroon niet gevonden
```

j = pointer

patroon op gelijke hoogte zetten met i vd tekst
volgorde karakters overeen komen volgend vgl'en
→ stop wnnr mis → i++ en opnieuw

↳ kan enorm veel werk zijn als we telkens op het einde een mismatch hebben of als tekst en patroon beide repetitief zijn

worst case telkens M testen op elke positie vh patroon

$M(N-M+1) = MN - MM + M \sim M \cdot N$ als $M \ll N$

↳ # startpos. patroon — karaktervergelijkingen

andere nadeel bruteforce: je moet minstens een deel van M bang vd tekst in geheugen houden voor als je merkt dat je terug naar startpos +1

back-up nodig

→ ~M extra geheugen

alternatieve implementatie - met back-up

```
public static int search (String pat, String txt) {
    int i , N = txt.length();
    int j , M = pat.length();
    for (i=0, j=0 ; i<N && j<M ; i++) {
        if (txt.charAt(i) == pat.charAt(j)) j++;
        else { i -= j ; j=0; }
    }
    if (j==M) return i-M;
    else      return N ; }
```

→ i is einde vh al gematchte ild tekst

→ j in # al gematchte karakters

→ back-up mechanisme

1) Knuth-Morris-Pratt (KMP)

back-up voor

gebruik maken van * kennis over hoe ver gevorderd
+ " " gefaalde match (mismatch)
+ " " structuur vh patroon

→ DETERMINISTISCHE

→ patroon wiskundig voorstellen als eindigetoestandsmachine (DFA)

⇒ tekst gewoon als input over DFA laten lopen en van toestand naar toestand springen totdat we in een aanvaarde eindtoest terecht komen of we door heel de tekst zijn gelopen (verworpen)

↳ overeenkomst met patroon → verder springen

mismatch → terug springen : niet telkens terug naar begintoest (multicast DFA)

langste # karakters vld prefix (van pat[I]) dat overeenkomt met de suffix (van txt[I])

→ naar die toestand DFA springen

toestand DFA komt overeen met # kar. al gematcht

DFA komt in een computer voor als een 2D-array
kolom → toestanden waarin je terecht kan komen
rij → mogelijke input (char.)
kolom → plaats in patroon } toestand waarin je terecht komt

```
public int search (String txt) {
    int i, j, N = txt.length();
    for (i=0, j=0; i<N && j<M; i++)
        j = dfa [txt.charAt(i)][j]
    if (j==M) return i-M;
    else      return N; }
```
} max N array accessen

geen backup nodig want maken niet terug naar vorige i
maken wel op voorhand de DFA hebben gemaakt / berekend

→ DFA opstellen: tabel invullen
- match transition
  stel in toestand j (= j dat al gematcht) en next char c == pat.charAt(j)
  → ga naar toestand j+1
- mismatch transition
  stel in toestand j en next char c != pat.charAt(j)
  → de laatste j karakters vlak input waren pat[1..j-1] gevolgd door c
  (want weten dat pat[0..j-1] = c toch niet gaat zijn)
  proberen deze nu door de DFA te laten lopen

recursief ⇒ blijft zich herhalen tot we een match tegenkomen

Analyse KMP
- karakter accessen: machine construeren (elk kar. patroon access.)
  + patroon door machine in tekst zoeken (lineair)
- dfa[][] opstellen: ⇒ max M+N kar.accesses
  tabel invullen = R (# kar. / grootte alfabet) x M (# toest / lengte pat.)
  → R·M getallen berekenen
  ⇒ tijd x geheugen ~ RM

2) Boyer-Moore

back-up nodig

check het patroon van achter naar voor: komt het laatste (M^e) kar vh patroon overeen met het kar. op pos M i/d tekst
- als mismatch
  * als dat kar. overeenkomt met een kar. uit het patroon
    → schuiven patroon zo op dat ze op dezelfde hoogte staan
  * als dat kar. niet overeenkomt met een kar uit het patroon (dubbele mismatch)
    → schuiven het patroon M pos. vooruit
- als match
  → vergelijk nu het voorlaatste karakter op dezelfde manier
  totdat je vanwaar bent beland in je patroon of op het einde van je tekst zit

↳ gebruiken hiervoor ile gegevensstructuur

nodig voor offset — right[c] → geeft meest rechtse voorkomen van kar. c i/h patroon terug
en -1 als het niet i/h patroon zit → positie
→ steeds updaten als je het karakter i/h patr. tegenkomt

```
public int search (String txt) {
    int N = txt.length(), M = pat.length();
    int skip;
    for (int i=0; i<=N-M; i+=skip) {
        skip=0;
        for (int j=M-1; j>=0; j--) {
            if (pat.charAt(j) != txt.charAt(i+j)) {
                skip = Math.max(1, j-right[txt.charAt(i+j)]);
                break; }
        }
match ---> if (skip==0) return i;
    }
    return N;
}
```

j: pointer die wijst naar de right[pat.charAt(j)] || van achter (M-1) naar voor

gemismatcht karakter uit de tekst
↳ meest rechtse voorkomen
om ervoor te zorgen dat we NOOIT terugspringen (altijd positieve skip)

analyse Boyer-Moore
worst case: telkens maar 1 pos opschuiven ⇒ terug in brute force situatie
~ M·N
best case: telkens M pos kunnen opschuiven
~ N/M
↳ kan combineren met KMP of andere hybrides maken om optitie te vermijden
→ worst case ~N

3) Rabin-Karp
hash berekenen en vergelijken beter en sneller dan het patroon karakter per karakter met de tekst te vergelijken

ipv karakters te vergelijken → hashcodes gebruiken
→ modulair hashen
stel R = #karakters i/h alfabet
⇒ patroon = getal in getallen systeem met basis R
van dit patroon de hash berekenen bv. 2 6535 % 997 = 613
voor elke i (i = 0, ..., N-M) berekenen we de hash van het getal gevormd door de kar. i t.e.m. i+M-1 in de tekst
als we hash patroon = hash tekst substring
→ mogelijke match, moeten checken of dit ook effectief een match is
↳ voordeel aan modulair hashen → kunnen modulo rekenen
⇒ incrementeel hash berekenen: gradueel hash updaten telkens dat we 1 stap opschuiven

$(a+b) \bmod Q = [(a \bmod Q) + (b \bmod Q)] \bmod Q$
$(a * b) \bmod Q = [(a \bmod Q) * (b \bmod Q)] \bmod Q$

aan de hand hiervan de hash vd patroon berekenen
eerst voor het 1e kar, 2e kar enzij, 3e kar ...
en de hash vd substrings i/d tekst
voor de eerste M: hash opbouwen zoals hash patroon
verder: steeds 1 getal vooraan weg x achteraan erbij
→ stel het M-digit getal met basis R als volgt voor

$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0$ met t_i = txt.charAt(i)

dan $x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \dots + t_{i+M} R^0$

$\Rightarrow x_{i+1} = x_i \cdot R - t_i \cdot R^M + t_{i+M}$

(shift left; subtract leftmost digit; add new rightmost digit)

$\Rightarrow x_{i+1} \bmod Q = \{[x_i \bmod Q + t_i \bmod Q \cdot (-R^{M-1} \bmod Q)](R \bmod Q) + t_{i+M} \bmod Q\}$
$= \{[x_i \bmod Q + t_i (Q - R^{M-1} \bmod Q)]R + t_{i+M}\} \bmod Q$ mod Q

(-a) mod Q = Q - a

x veronderstel dat $R < Q, t_i < Q \Rightarrow R \bmod Q = R$ x $t_i \bmod Q = t_i$

2 varianten Rabin
- Las Vegas variant
  - wel back-up nodig
  - als de hash gelijk is → met patroon karakter per karakter nog vergelijken met de substring
  - Grootte ~M
  - 100% correct
  - Vaakste M kar txt nog vgl'en
  - heel waarschijnlijk lineaire tijd ~N
  - worst case echter ~M·N
- Monte Carlo variant
  - geen back-up nodig
  - gaan ervan uit dat als hash gelijk → patroon v substring ook gelijk
  - kans op mismatch = 1/Q → geen 100% zekerheid, maar zekerheid groter wnnr Q groter
  - altijd in lineaire tijd ~N

? modulo berekenen trager dan één karakter compar maar kan wel meerdere pat herkennen

Slides Dynamic programming

Dynamic programming

→ manier van algoritmes implementeren / ontwerpen
= look-up tabel (berekeningen in tabellen simuleren, cfr Excel)
info omtrent v/h algoritme tijdelijk opslaan (tussenresultaten tijdelijk opslaan)
om deze later te hergebruiken om complexe recursieoproepen te vermijden

basisidee
probleem opdelen in subproblemen
de optimale opl. voor deze subproblemen → optimale opl voor het globale probleem
deze subproblemen delen ook dezelfde subsubproblemen, enz.
└ normaal: recursie → opdelen tot in triv. geval → sub... subproblemen
MAAR echter: veel van die subsubproblemen zijn identiek
⇒ ipv steeds hetzelfde op te lossen, gaan we die nu 1 keer oplossen en het resultaat in een tabel opslaan om te kunnen hergebruiken
→ sub...subprobleem dat we herkennen opzoeken i/d tabel
⇒ opl. blijft hetzelfde maar efficiënter op tijdscomplexiteitsvlak

vb rij van Fibonacci
recursief programma

```
public static int fibonacci(int n) {
    int fib;
    if (n == 0)  fib = 0;
    else if (n == 1) fib = 1;
    else fib = fibonacci(n-1) + fibonacci(n-2);
    return fib; }
```

$T(0) = 1$
$T(1) = 1$
$T(n) = T(n-1) + T(n-2)$
$\Rightarrow T(n) \geq 2T(n-2) = 2^{n/2}$

└ werkt, maar héél tijdsineff. want doen veel dubbel werk (vnl wnnr hogere n)
⇒ beter: bottom-up

```
public static int fibonacci(int n) {
    int[] fib = new int[n];
    fib[0] = 1, fib[1] = 1;
    for (i = 2; i <= n; i++)
        fib[i] = fib[i-1] + fib[i-2];
    return fib[n]; }
```

lineaire tabel die we plaats per plaats invullen
→ maken i elementen berekenen
⇒ lineaire tijd, $\sim n$

vermijden recursieve oproepen die dubbel uitgevoerd worden en volgend dezelfde resultaten te laten afhangen vd vorige

vb staaf cutting: staaf in stukken snijden (sprongen van 1 blokje) meer a/h v/h # blokjes waaruit het stuk bestaat, heeft elk stuk een bep. waarde willen de waarde vd staaf totaal zo hoog mogelijk hebben

naïeve opl: staaf in 2 delen, alle $n-1$ mogelijke verdelingen uitvoerend en dan recursief het overgebleven stuk weer volgens alle mogelijke verdelingen in 2 delen → zo recursief verder doen
└ echter 2^{n-1} mogelijkheden die, vooral naarmate de staaf korter wordt, steeds beginnen overlappen met andere situaties
→ exponentieel

top-down: tabel bijhouden die de best waarde aanduidt voor het verdelen vd staaf van bep. lengte
1e keer staaf van die lengte? → recursief berekenen zoals hierboven, dan resultaat opslaan i/d tabel om later gewoon te kunnen opzoeken
→ kwadratisch

bottom-up: gelijkaardig aan fibonacci
eerst waarde van staaf van lengte 1 berekenen, aan de hand daarvan staaf van lengte 2 → herhalen / opbouwen tot lengte n
└ zelf ingrijpen, geen recursie, gewoon invullen
→ kwadratisch

vb rij munten (even aantal) met elk een waarde
spelers mogen afwisselend een munt nemen, maar telkens enkel de uiterst rechtse of linkse munt (2 spelers)
speler die op het einde de hoogste waarde heeft, wint
→ greedy algoritme? Neen – want dan je zo een munt met hogere waarde voor je tegenspeler wegspeelt

→ opdelen in subproblemen
- nummer alle munten van 1 t.e.m. n
- houd de waarde bij i/e array: V[1] = ..., V[2] = ..., ..., V[n] = ...
→ voor een queuen vrij van munt i t.e.m. munt j
kiest speler 1 ofwel V[i] ofwel V[j]
wat is nu de maximale waarde die je uit de reeks i..j kan halen als we het spel (met 2 spelers die elk optimaal spelen,) uitspelen? M_{ij}?

$M_{i,i+1} = \max(V[i], V[i+1])$ voor $j = i+1$: 2 munten over → kies de grootste van V[i] en V[j]
voor $j > i+1$: meer dan 2 munten over

$$M_{ij} = \max\{ \min(M_{i+1,j-1}, M_{i+2,j}) + V[i],\ \min(M_{i,j-2}, M_{i+1,j-1}) + V[j] \}$$

- speler 1 kiest munt i en speler 2 dan j of i+1 → min want speler 2 wil ook optimale voor zichzelf
- speler 1 kiest munt j en speler 2 j-1 of i → min want speler 2 wil ook optimale voor zichzelf

⇒ tabel opstellen voor M_{1n} uit te halen
aangezien een # munten → j-i+1 in een

i ↓ \ j →	1	2	3	4	5	6
1	6		9	9	15	15
2		5	9	10	9	
3			7	7	13	
4				7	7	
5					5	
6						

voor 1 5 2 7 3 5 / 2 3 4 5 6

- initialisatie : $j = i+1 \rightarrow j-i+1 = 2$
invullen met $\max[V[i], V[i+1]]$
aangezien $i<j$ → moeten enkel bovenste helft invullen
- vul M_{ij} in voor alle $j-i+1 = 4$
- " " " " " $j-i+1 = 6$
→ herhalen tot M_{1n} bereikt

opvullen ⇒ kwadratische tijd vs exponentieel van puur recursief uitwerken
$\sim n^2/4$ want maken door dynamic progr. maar deel invullen

Longest Common Subsequence (LCS) !

hoe gelijkaardig zijn 2 (lange) strings van karakters?
→ gemeenschappelijke subsequenties = stukjes van karakters in de strings
(≠ substring!) die op elkaar volgen (zelfde volgorde, relatief)
onderlinge volgorde, → maar niet noodzakelijk exact achter elkaar staan
→ $X[i_1]X[i_2]\dots X[i_{k-1}]X[i_k]$ met $i_j < i_{j+1}$ $j = 1, \dots, k-1$

→ langst gemeenschappelijke subsequentie
- naïeve opl : alle mogelijke subseq van string X (lengte n) opsommen
→ 2^n mogelijke subseq
die nu checken of ze ook een subseq zijn van string Y (lengte m)
→ ~ m tijd voor 1 subseq te checken (lineair door strings lopen)
⇒ totaal ~ $2^n m$ (exp)
+ heel veel overlappende problemen want veel van de 2^n subseq overlappen

⇒ dynamic programming

dynamic programming om LCS op te lossen

string X met lengte m, string Y met lengte m

→ subprobleem: langst mogelijke gemeensch. subseq in DELEN van X en Y

→ LCS van X[0..i] en Y[0..j] ≡ L[i,j]

→ voorstellen i/e matrix en uiteindelijk L[m-1, n-1] berekenen

* Stel X[i] = Y[j] = c

⇒ LCS van X[0..i] en Y[0..j] eindigt met karakter c

Bewijs uit het ongerijmde

Stel L[i,j] eindigt op een ander karakter dan c, dan kunnen we L[i,j] uitbreiden met c want X eindigt en Y ook → tegenspraak

Stel L[i,j] eindigt met een c, maar die c in X en/of Y zit op een andere pos dan i,j. Dan kn we de pos. van c forceren naar de laatste pos van X en Y. □

als X[i] = Y[j] → L[i,j] gereduceerd tot korter probleem

⇒ L[i,j] = L[i-1, j-1] + 1 als X[i] = Y[j] → allebei korter

* Stel X[i] ≠ Y[j]

als X[i] ≠ Y[i] → strings telkens eentje knippen en max LCS bepalen

⇒ LCS kan niet tegelijk op X[i] en Y[j] eindigen,
LCS kan op X[i], Y[i] of geen van beiden eindigen

⇒ L[i,j] = max{L[i-1, j], L[i, j-1]} als X[i] ≠ Y[i]

knippen in X / knippen in Y → zien welke vld 2 LCS nu mogelijkste is

* Randgevallen: L[i,-1] = L[-1,j] = 0 → -1 wijst op lege string

→ Algoritme: tabel L[] opvullen beginnend bij L[0,0] en zo verder werken totdat we L[m-1, n-1] hebben bereikt

~ n·m

L	-1	0	1	2	3	4	5	→ j (Y)
-1	0	0	0	0	0	0	0	
0	0	0	1	1	1	1	1	
1	0	0	1	1	2	2	2	
2	0	0	1	2	2	3	3	
3	0	1	1	2	2	2	2	
↓ i (X)								

Y = CGATAA
X = GTAC

LCS is niet uniek, maar de lengte ervan wel

○ duidt pos. aan waarvoor de eindkar hetzelfde zijn

1) rijen en kolommen met index -1 ingevuld met 0
2) invullen v/n/z en v/b/n/o
   X[0]Y[0] → gelijk ⇒ 1, niet gelijk ⇒ 0
   X[0]Y[1] verder aanvullen zoals hierboventem X[0]Y[m-1]
   zodra ergens in Y: Y[j] = X[0] → verander naar 1 en blijft voor de rest ook staan want blijft in opvolgende Y zitten
3) gelijk (○) → diagonaal + 1
   verschil → max(links, boven)

- Optimale binaire zoekbomen ≠ gebalanceerde binaire boom

voor een gegeven set $k_1 < k_2 < \dots < k_n$ van n gesorteerde sleutels (elke k_i met waarsch. p_i)
wat is nu de BST met minimum verwachte zoekkost?

$kost_{k_i}$ = # vgl'en nodig om k_i te vinden
= diepte(k_i) + 1 → wortel op diepte 0

↳ gemiddelde kost voor k_i = kost i vgl m frequentie/kans voorkomen

$kost_{k_i, gem} = (diepte(k_i) + 1) \cdot p_i$

⇒ totale kost vld zoekboom

$$kost_{BST} = \sum_{i=1}^{n} (diepte(k_i) + 1) \cdot p_i$$

$$= \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} diepte(k_i) \cdot p_i$$

→ optimale zoekboom heeft kleinste kost
- niet per se kleinste lengte
- niet per se meest freq. bij als wortel

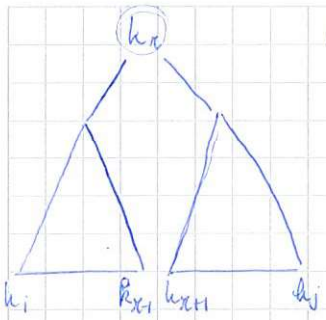
vinden adv dynamic programming

- optimale zoekboom vinden voor $k_i, \dots, k_j$ $(1 \le i \le j \le n)$
  → telkens uitbreiden door 2 reeds berekende optimale zoekbomen te combineren
  aangezien voor optimale BST T geldt dat als T subboom T' met $k_i, \dots, k_j$ bevat, dan is T' ook optimale BST voor die keys
- → optimale kost voor boom met $k_i, \dots, k_j \equiv C(i,j)$

tabel invullen met als $i = j \Rightarrow C(i,j) = 1$

$j < i \Rightarrow C(i,j) = 0$

$j > i \Rightarrow C(i,j) = p_r + C(i, r-1) + W(i, r-1) + C(r+1, j) + W(r+1, j)$



$i \leq r \leq j$

stel dat k_r de wortel is vd optimale BST voor $k_i, \dots, k_j$
dan hebben we links een subboom van $k_i, \dots, k_{r-1}$
en rechts van $k_{r+1}, \dots, k_j$
elk met hun eigen optimale kost, $c(i, r-1)$ en $c(r+1, j)$
door het combineren vd 2 bomen
- diepte van alle knopen in vd subboom met 1 verhoogt
  → kost subboom vermeerdert met de som van alle waarsch. vd subboom, zeg w

$\Rightarrow c(i,j) = p_r + c(i,r-1) + w(i,r-1) + c(r+1,j) + w(r+1,j)$

(p_r: kost k_r; $w(i,r-1)$: som waarsch linkersubboom)

en aangezien $w(i,j) = w(i,r-1) + p_r + w(r+1,j)$

$\Rightarrow c(i,j) = p_r + c(i,r-1) + w(i,r-1) + c(r+1,j) + w(r+1,j)$
$= c(i,r-1) + c(r+1,j) + w(i,j)$

→ optim: $c(i,j) = \min_{i \leq r \leq j} \{ c(i,r-1) + c(r+1,j) + w(i,j) \}$

maken 3 2D-arrays berekenen
- $c[1 \dots n, 1 \dots n]$ → kost opslaan
- $w[1 \dots n, 1 \dots n]$ → w niet telkens opnieuw willen berekenen $w[i,j] = w[i,j-1] + p_j$
- $root[i,j]$ → optimale r voor subboom $i \dots j$

→ eindresultaat: $c[1,n]$ en $root[1,n]$