Samenvatting G&A

1 Les 2: Basic Sorting Methods

1.1 Selection Sort

Algoritme 1: Selection Sort

```
public class Selection{
    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;
            exch(a, i, min);
        }
    }
}
```

Proof 1: Selection Sort uses $\sim N^2/2$ compares and N exchanges to sort an array of length N.
This can be proven by examining the trace, which is an N-by-N table in which unshaded letters correspond to compares. About one-half of the entries in the table are unshaded - those on and above the diagonal. The entries on the diagonal each correspond to an exchange. More precisely, examination of the code reveals that, for each i from 0 to N-1, there is one exchange and N - 1 - i compares, so the totals are N exchanges and $(N-1)+(N-2)+...+2+1+0 = N(N-1)/2 \sim N^2/2$ compares.

1.2 Insertion Sort

Algoritme 2: Insertion Sort

```
public class Insertion{
    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--)
                exch(a, j, j-1);
        }
    }
}
```

Proof 2: Insertion Sort uses $\sim N^2/4$ compares and $\sim N^2/4$ exchanges to sort a randomly ordered array of length N with distinct keys, on the average. The worst case is $\sim N^2/2$ exchanges and the best case is $N-1$ compares and 0 exchanges.
Like in Proof 1, this can e visualized in the N-by-N diagram we use to illustrate the sort. We count entries below the diagonal - all of them, in the worst case, and none of them, in the best case. For randomly ordered arrays, we expect each item to go about halfway back, on the average, so we count one-half of the entries below the diagonal. The number of compares is the number of exchanges plus an additional term equal to N minus the numer of times the item is inserted is the smallest so far. In the worst case (array in reverse order), this term is negligible in relation to the total; in the base case (array is in order) it is equal to N - 1.

1.3 Merge Sort

Algoritme 3: Merge Sort

```
1  public class Merge{
2      private static Comparable[] aux;
3
4      public static void sort(comparable[] a){
5          aux = new Comparable[a.length];
6          sort(a, aux, 0, a.length - 1);
7      }
8
9      public static void sort(Comparable[] a, Comparable[] aux,
10                                              int lo, int hi){
11         if(hi <= lo) return;
12         int mid = lo + (hi - lo) / 2;
13         sort(a, aux, lo, mid);
14         sort(a, aux, mid + 1, hi);
15         merge(a, aux, lo, mid, hi);
16     }
17
18     public static void merge(Comparable[] a, int lo, int mid, int hi){
19         Comparable aux = new Comparable[a.length];
20         for (int k = lo; k <= hi; k++) aux[k] = a[k]; //copy
21         int i = lo, j = mid + 1;                      //merge
22         for (int k = lo; k <= hi; k++) {
23             if(i > mid)                    a[k] = aux[j++];
24             else if(j > hi)                a[k] = aux[i++];
25             else if(less(aux[j], aux[i]))  a[k] = aux[j++];
26             else                           a[k] = aux[i++];
27         }
28     }
29 }
```

Proof 3: Merge Sort uses between $\sim 1/2Nlog_2(N)$ and $\sim Nlog_2(N)$ compares and exchanges to sort an array of length N.
Let C(N) be the number of compares needed to sort an array of length N. We have C(0) = C(1) = 0 and for N > 0 we can write a recurrence relationship that directly mirrors the recursive sort() method to establish an upper bound: $C(N) \leq C([N/2]) + C([N/2]) + N$. The first term on the right is the number of compares to sort the left half of the array, the second term is the number of compares to sort the right half, and the third term is the number of compares for the merge. The lower bound $C(N) \geq C([N/2]) + C([N/2]) + N$ follows because the number of compares for the merge is at least [N/2]. We derive an exact solution to the recurrence when equality holds and N is a power of 2. First, since [N/2] = 2^{n-1}, we have C(2^n) = 2C(2^{n-1}) + 2^n. We divide both sides by 2^n and apply the same equation to the first term on the right to get $C(2^n)/2^n = C(2^{n-2})/2^{n-2} + 1 + 1$. If we repeat the previous step n-1 additional times we get $C(2^n)/2^n = C(2^0)/2^0 + n$, which, after multiplying both sides by 2^n leaves us with the solution: C(N) = C(2^n) = n2^n = $Nlog_2(N)$
Note: Merge Sort does not work in place, unlike the previous two sorting algorithms. It uses more space than N.

1.4 Addendum: benadering van Stirling

Te bewijzen: $log_2(n!) = nlog_2(n)$
Bewijs: $log_2(n!) = log_2(e)ln(n!)$
$= log_2(e)[ln(n) + ln(n-1) + ... + ln(2) + ln(1)]$
We benaderen vervolgens de som van de natuurlijke logaritmes door een integraal (wat een kleine benaderingsfout als gevolg heeft).
$\approx log_2(e) \int_1^n ln(x)dx$
$= log_2(e)[nln(n) - n + 1]$
$= nlog_2(n) - nlog_2(e) + log_2(e)$
$\approx nlog_2(n)$

2 Les 3 & 4: QuickSort

The crux of the method is the partitioning process, which rearranges the array to make the following three conditions hold:

- The entry a[j] is in its final place in the array, for some j.
- No entry in a[lo] through a[j-1] is greater than a[j].
- No entry in a[j+1] through a[hi] is less than a[j].

2.1 QuickSort variant 1

→ Choose pivot element (e.g. first element)
→ Create 2 arrays: *left[]* and *right[]*
→ Loop over all elements
  → If element < pivot, put into *left[]*
  → If element > pivot, put into *right[]*
  → (if element == pivot, left or right)
→ Concatenate *left[]* + pivot + *right[]*

2.2 QuickSort variant 2

Algoritme 1: QuickSort (variant 2)

```
1  public class Quick{
2      public static void sort(Comparable[] a){
3          StdRandom.shuffle(a);      //Eliminate dependence on input
4          sort(a, 0, a.length - 1);
5      }
6
7      private static void sort(Comparable[] a, int lo, int hi){
8          if(hi <= lo) return;
9          int j = partition(a, lo, hi);
10         sort(a, lo, j-1);
11         sort(a, j+1, hi);
12     }
13
14     private static int partition(Comparable[] a, int ll, int hi){
15         int i = lo, j = hi + 1; //left and right scan indices
16         Comparable v = a[lo];   //partitioning item
17         while (true){
18             //scan right, scan left, check for scan complete, and exchange
19             while (less(a[++i], v)) if (i == hi) break;
20             while (less(v, a[--j])) if (j == lo) break;
21             if (i >= j) break;
22             exch(a, i, j);
23         }
24         exch(a, lo, j); //Put partitioning item v into a[j]
25         return j;       //with a[lo ... j-1] <= a[j] <= a[j+1 ... hi]
26     }
27 }
```

2.3 Variant 3: Lumoto

Algoritme 2: QuickSort (Lumoto)

```
1  public class Quick{
2     private static int partition(Comparable[] a, int lo, int hi){
3         // Partition into a[lo..i], a[i+1], a[i+1..hi].
4          int i = lo-1; // scan index
5          Comparable v = a[hi]; // partitioning value
6          for (int j = lo; j <= hi-1; j++){
7           // scan all elements
8              if (less(a[j],v)) {
9                  i++;
10                 exch(a, i, j);
11             }
12         }
13         exch(a, i+1, hi); // Put v into position
14         return i+1; // with a[lo..i] <= a[i+1] <= a[i+1..hi].
15     }
16 }
```

2.4 Tijdscomplexiteit

Proof 1: QuickSort uses $1.39Nlog_2(N)$ comparisons on average.
Let C be the number of comparisons and N the number of elements in the array. The precise recurrence satisfies $C_0 = C_1 = 0$ and for $N \geq 2$:
$C_N = N + 1 + ((C_0 + C_{N-1}) + ... + (C_{k-1} + C_{N-k}) + ... + (C_{N-1} + C_1))/N$
$= N + 1 + 2(C_0 + ... + C_{k-1} + ... + C_{N-1})/N$
We multiply both sides by N:
$NC_N = N(N + 1) + 2(C_0 + ... + C_{k-1} + ... + C_{N-1})$
We subtract the same formula for N-1:
$NC_N - (N_1)C_{N_1} = N(N + 1) - (N_1)N + 2C_{N-1}$
Next, we simplify:
$NC_N = (N + 1)C_{N_1} + 2N$
We divide both sides by N(N+1) to get a telescoping sum:
$C_N/(N + 1) = C_{N_1}/N + 2/(N + 1)$
$= C_{N-2}/(N - 1) + 2/N + 2/(N + 1)$
$= C_{N-3}/(N - 2) + 2/(N - 1) + 2/N + 2/(N + 1)$
$= 2(1 + 1/2 + 1/3 + ... + 1/N + 1/(N + 1))$
We approximate the exact answer by an integral:
$C_N \approx 2(N + 1)(1 + 1/2 + 1/3 + ... + 1/N)$
$\approx 2(N + 1) \int_1^n dx/x$
Finally, we get the desired result:
$C_N \approx 2(N + 1)lnN$
$\approx 1.39Nlog_2(N)$

Proof 2: QuickSort uses $\sim N^2/2$ compares in the worst case.
Note: random shuffling of the array protects against this case.
The number of compares used when one of the subarrays is empty for every partition is:
N + (N-1) + (N-2) + ... + 2 + 1 = (N+1) N/2 $\sim N^2/2$

2.5 Optimalisaties

→ Cutoff to insertion sort
  → ∼ 10 elements
→ Median of 3 values for pivot
  → Better probability of splitting roughly in half
→ Many similar values
  → 3-way partitioning (less, equal, greater)

Algoritme 3: 3-way QuickSort

```
1  public class Quick3way{
2      private static void sort(Comparable[] a, int lo, int hi){
3          if (hi<=lo) return;
4          int lt = lo, i = lo+1, gt = hi;
5          Comparable v = a[lo];
6          while (i <= gt){
7              int cmp = a[i].comparaTo(v);
8              if      (cmp < 0) exch(a, lt++, i++);
9              else if (cmp > 0) exch(a, i, gt--);
10             else              i++;
11         }
12         sort(a, lo, lt-1);
13         sort(a, gt + 1, hi);
14     }
15 }
```

3 Les 5: Sorting in linear time

3.1 Key-indexed counting

Algoritme 1: key-indexed counting

```
1  public class Key{
2      private static void sort(Item[] a, int R){
3          Item[] aux = new String[N];
4          int[] count =  new int[R+1];
5
6          //Compute frequency counts
7          for (int i = 0; i < N; i++)
8              count[a[i].key() + 1]++;
9          //Transform counts to indices
10         for (int r = 0; r < R; r++)
11             count[r+1] += count[r];
12         //Distribute the records
13         for (int i = 0; i < N; i++)
14             aux[count[a[i].key()]++] = a[i];
15         //Copy back
16         for (int i = 0; i < N; i++)
17             a[i] = aux[i];
18     }
19 }
```

Key-indexed counting wordt gebruikt op data waarbij de keys kleine integers zijn. Eerst wordt voor elke key opgeteld hoeveel keer deze voorkomt in de array. Vervolgens worden deze aantallen omgezet naar indices: stel dat key[1] 5 keer voor kwam, key[2] 3 keer voor kwam en key[3] 7 keer voor kwam, dan begint key[1] met index 0, key[2] met index 5 en key[3] op index 8 (en key[4] dan op index 15). Vervolgens wordt dit dan in een auxiliary array gestopt. Hierdoor is het algoritme stabiel: de volgorde bij elementen met dezelfde key wordt behouden.

Proof 1: Key-indexed counting uses 11N + 4R + 1 array accesses to stably sort N items whose keys are integers between 0 and R-1.

Initializing the arrays uses N + R + 1 accesses. The first loop increments a counter for each of the N items (3N accesses); the second loop does R additions (3R array access); the third loop does N counter increments and N data moves (5N array accesses); and the fourth loop does N data moves (2N array accesses).

3.2 Least-significant digit first (radix)

Algoritme 2: LSD string sort

```
1  public class LSD{
2      public static void sort(String[] a, int R){
3          int N = a.length;
4          int R = 256;
5          String[] aux = new String[N];
6
7          for (int d = W-1; d >= 0; d--){
8              //Compute frequency counts
9              int[] count = new int[R+1];
10             for (int i = 0; i < N; i++)
11                 count[a[i].charAt(d) + 1]++;
12             //Transform counts to indices
13             for(int r = 0; r < R; r++)
14                 count[r+1] += count[r];
15             //Distribute
16             for (int i = 0; i < N; i++)
17                 aux[count[a[i].charAt(d)]++] = a[i];
18             //Copy back
19             for (int i = 0; i < N; i++)
20                 a[i] = aux[i];
21         }
22     }
23 }
```

To sort an array a[] of strings that each have exactly W characters, we do W key-indexed counting sorts: one for each character positions, proceeding from right to left. We make use of the fact that key-indexed counting sort is stable, because after each iteration the order is kept among the elements.

Proof 2: LSD string sort uses $\sim 7WN + 3WR$ array accesses and extra space proportiaonal to N + R to sort N items whose keys are W-character strings taken from an R-character alphabet.

The method is W passes of key-indexed counting, except that the aux[] array is initialized just once. The total is immediate from the code and proof 1.

3.3 Most-significant digit first

Algoritme 3: MSD string sort

```
1  public class MSD{
2      private static int R = 256;       //radix
3      private static final int M = 15;
4      private static String[] aux;
5
6      private static int charAt(String s, int d){
7          if(d < s.length()) return s.charAt(d); else return -1;}
8
9      public static void sort(String[] a){
10         int N = a.length;
11         aux = new String[N];
12         sort(a, 0, N-1, 0);
13     }
14
15     private static void sort(String[] a, int lo, int hi, int d){
16         //Cutoff for small subarrays
17         if (hi <= lo + M){
18             Insertion.sort(a, lo, hi, d); return;}
19         //Compute frequency counts
20         int[] count = new int[R+2];
21         for (int i = lo; i <= hi; i++)
22             count[charAt(a[i], d) + 2]++;
23         //Transform counts to indices
24         for (int r = 0; r < R+1; r++)
25             count[r+1] += count[r];
26         //Distribute
27         for (int i = lo; i <= hi; i++)
28             aux[count[charAt(a[i], d) + 1]++] = a[i];
29         //Copy back
30         for(int i = lo; i <= hi; i++)
31             a[i] = aux[i - lo];
32         //Recursively sort for each character value
33         for (int r = 0; r < R; r++)
34             sort(a, lo + count[r], lo + count[r+1] - 1, d+1);
35     }
36 }
```

The basic idea behind MSD string sort is that in typical applications, the strings will be in order after examining only a few characters in the key. The method quickly divides the array to be sorted into small subarrays. Because we have a huge number of tiny subarrays, we must make sure to sort them efficiently. If we do not use a cutoff, we will have 258 (on an ASCII string where R=256) entries in the count[] array for every sort of a subarray of length 1. With unicode (R=65536) this might be thousands of times slower. Thus, we use insertion sort for small subarrays. *This is a must for MSD string sort.*

Proof 3: MSD string sort uses between 8N +3R and $\sim 7wN + 3wR$ array accesses to sort N strings taken from an R-character alphabet, where w is the average string length. The amount of space needed is proportional to R times the length of the longest string (plus N) in the worst case.

In the best case, MSD sort uses just one pass; in the worst case, it performs like LSD string sort. The count[] array must be created within sort(), so the total amount of space needed is proportional to R times the depth of recursion (plus N for the auxiliary array).

3.4 Three-way string quicksort

Algoritme 4: Three-way string quicksort

```
1  public class Quick3string{
2      private static int charAt(String s, int d){
3          if(d < s.length()) return s.charAt(d); else return -1;}
4
5      public static void sort(String[] a){
6          sort(a, 0, a.length-1, 0);}
7
8      private static void sort(String[] a, int lo, int hi, int d){
9          //Cutoff for small subarrays
10         if (hi <= lo)
11             return;
12         int lt = lo, gt = hi;
13         int v = charAt(a[lo], d);
14         int i = lo + 1;
15         while(i <= gt){
16             int t = charAt(a[i], d);
17             if      (t < v) exch(a, lt++, i++);
18             else if (t > v) exch(a, i, gt--);
19             else            i++;
20         }
21
22         sort(a, lo, lt-1, d);
23         if (v >= 0) sort(a, lt, gt, d+1);
24         sort(a, gt+1, hi, d);
25     }
26 }
```

Three-way string quicksort divides the array into only three parts, so it involves more data movement than MSD string sort when the number of nonempty partitions is large because it has to do a series of 3-way partitions to get the effect of the multiway partition. On the other hand, MSD string sort can create large numbers of (empty) sub-arrays, whereas quicksort always has just three thus quicksort adapts well to handling equal keys, keys with long common prefixes, keys that fall into a small range, and small arrays - all situations where MSD string sort runs slowly. Three-way quicksort also needs no additional space other than the implicit stack to support recursion unlike MSD which requires space for frequency counts and an auxiliary array.

Proof 4: 3-way quicksort uses $\sim 2Nln(N)$ character compares on the average to sort an array of N random strings

First, considering the method to be equivalent to quicksort partitioning on the leading character, then (recursively) using the same method on the subarrays, we should not be surprised that the total number of operations is about the same as for normal quicksort - but they are single-character compares, not full-key compares. Second, considering the method as replacing key-indexed counting by quicksort, we expect that the $Nlog_R(N)$ running time from proof 3 should be multiplied by a factor of 2 ln(R) because it takes quicksort 2Rln(R) stepts to sort R characters, as opposed to R steps for the same characters in the MSD string sort.

3.5 Bucket Sort

Algoritme 5: Bucket sort

```
1  public class Bucket{
2      public sort(Item[] a, int k){
3          int N = a.length;
4          Item[] buckets = new Item[k]
5
6          //Divide elements into buckets
7          for(int i = 0; i < N; i++)
8              buckets[a[i]/max(a)].append(a[i]);
9
10         //Sort buckets
11         for(int = 0; i < k; i++)
12             Insertion.sort(buckets[i]);
13
14         //Concatenate buckets
15         for(int i = 0; i < k; i++)
16             for(int j = 0; j < buckets[i].length; j++)
17                 a[i*j + j] = buckets[i][j];
18     }
19 }
```

Bucket sort divides the input range (n elements) into k buckets, and then uses a comparison-based sort on each bucket. In the worst case, everything gets put into one bucket, giving us no increase in performance. In the best case, the input is divided evenly over the bucket, reducing the problem size for each bucket to n/k + $\sim n$. On the average case, if we assume uniform random input, we will get the same results as the best case. Bucket sort does have some tradeoffs:
→ Only works well for evenly distributed inputs
  → Choose buckets well
→ Requires $\sim n$ extra space
  → n elements divided over k buckets plus the original array

3.6 Examenvragen

Een aantal sorteeralgoritmes hebben de eigenschap "stabiel"te zijn.

- **Wat wordt hiermee bedoeld?**
  Als verschillende elementen in een array dezelfde key hebben, zullen ze in de gesorteerde array in dezelfde volgorde staan.
- **Welke sorteeralgoritmes die we behandeld hebben in de lessen zijn stabiel?**
  Key-indexed counting,
- **Waarom is stabiliteit een belangrijke eigenschap van een sorteeralgoritme?**
- **Kan een niet-stabiel sorteeralgoritme steeds stabiel gemaakt worden?**

Hoe zou je counting sort kunnen wijigen zodat het ook zou werken met data die geen positieve integers zijn? Bvb ook negatieve integers behandelen? Of ook getallen met decimale fracties (1.5, 2.5, 3.5, ... ?). Welke voorwaarden moeten er gelden voor de te sorteren data?

4 Les 6: Stacks, Queues & Hash tabellen

4.1 Stacks

→ Remove the item the most recently added.
Implemented as a linked list; pop() and push() run in constant time. It can also be implemented with an array: but then a problem arrises: what to do when the array is full. We must then make a new, bigger array and copy all item to that new array (big cost!). When we always increase the array size by 1, we get a cost of $\sim N^2 (= N + 2(1 + 2 + ... + N - 1)$. Instead, we double the size when the array is full (cost is $\sim 3N (= N + (1 + 2 + 4 + 8 + .. + N/2))$. When the array is only 1/4th full, we halve the array size (so that the array is always between 25% and 100% full is).
Note: This protects against cases where you use "pop-push-pop-push-...".
→ Linked list implementation
  → Every implementation takes constant time in the worst case
  → Uses extra time and space to deal with the links
→ Dynamic array implementation
  → Every operation takes constant amortized time
  → Less wasted space

4.2 Queues

→ Remove the item the least recently added.
Queues use the same principles as stacks and can also be implemented with either a linked list or a dynamic array.

4.3 Hash tabellen

4.3.1 Keys omzetten naar indices

De bedoeling van een hashfunctie is dat ze een sleutel kan omzetten naar een index voor de array. Gegeven een array van lengte M moet onze hash-functie dus een integer in het bereik [0, M-1] teruggeven. De meest gebruikte methode is modulair hashen: we kiezen M opdat dit een priemgetal is en onze hash wordt dan berekend door de modulo van k/M te berekenen: k % M. Strings worden bekeken als grote integers als hun hashcode berekend wordt. Er zijn echter nog wel problemen met hashing.
Problem 1: No deterministic hash function can uniformly and independently distribute keys among the integer values between 0 and M-1.

4.3.2 Hashing with seperate chaining

Hashing met seperate chaining is een manier om hash collisions op te lossen: wanneer twee waarden dezelfde hash hebben. Bij seperate chaining worden de botsende elementen aan elkaar gelinkt. Als M groot genoeg is, is onze lijst steeds klein genoeg om geen performance overhead te hebben. Als we N keys hebben, zal elke lijst gemiddeld N/M elementen groot zijn.

4.3.3 Hashing with lineair probing

Another approach to implementing hashing is to store N key-value pairs in a hash table of size M > N, relying on empty entries in the table to help with collision resolution. These mehtods are called open-addressing hashing methods. Linear probings checks the next entry in the table if there is a collision (by incrementing the index). Linear probing is characterized by identifying three possible outcomes:

- Key equal to search key: search hit
- Empty position (null key at indexed position): search miss
- Key not eaqual to search key: try next entry

5 Les 7: Priority Queues & Balanced Trees

5.1 Priority Queues

Priority Queus are queues that return the element with the highest (or lowest) priority in the queue. These can be implemented in a number of ways, but we will discuss a binary heap. A binary heap is a heap-ordered binary tree. (Note: a binary tree has the property that the height of a tree with N nodes is always $1 + log_2(N)$). In the binary heap, keys are stored in nodes. The key in a node is never smaller than the keys in the children. If we use an array representation, we take the nodes in level order and do not need explicit links. We use the array indices to move through a tree: the parent of node k is at k/2 and the children of node k are at 2k and 2k+1. (We assume that the largest key is at a[1].) If a node's key is replaced with a key larger than its parent, we simply swap them until order is restored. Insertion and deletion follow the same principles. They use at most $2log_2(N)$ comparisons.
Building a heap uses $\sim Nlog_2(N)$ comparisons ($= log_2(1) + log_2(2) + ... + log_2(N)$ cfr stirling approximation).

Algoritme 1: Heap Priority Queue

```
1  public class MaxPQ<Key extends Comparable<Key>>{
2      private Key[] pq;
3      private int N = 0;
4
5      public MaxPQ(int maxN){
6          pq = (Key[]) new Comparable[maxN+1];}
7
8      public boolean isEmpty(){
9          return N == 0;}
10
11     public int size(){
12         return N;}
13
14     public void insert(Key v){
15         pq[++N] = v;
16         swim(N);
17     }
18
19     public Key delMax(){
20         Key max = pq[1];     //Retrieve max key from top
21         exch(1, N--);        //Exchange with last item
22         pq[N+1] = null;      //Avoid loitering
23         sink(1);             //Restore heap property
24         return max;
25     }
26
27     private void swim(int k){
28         while (k > 1 && less(k/2, k)){
29             exch(k/2, k);
30             k = k/2;
31         }
32     }
33
34     private void sink (int k){
35         while (2*k <= N){
36             int j = 2*k;
37             if (j < N && less(i, j+1)) j++;
38             if(!less(k, j)) break;
39             exch(k, j);
40             k = j;
41         }
42     }
43 }
```

5.2 Binary and Balanced Search Trees (BSTs)

5.2.1 Binary Search Trees

Proof 1: Search hits in a BST built from N random keys require $1.39Nlog_2(N)$ compares on average.
The number of compares used for a search hit eding at a given node is 1 plus the depth. Adding the

depths of all nodes, we get a quantity known as the internal path length of the tree. Thus, the desired quantity is 1 plus the average internal path length of the BST. Let C_N be the internal path length of a BST built from inserting N randomly ordered distinct keys, so that the average cost of a search hit is $1 + C_N/N$. We have $C_0 = C_1 = 0$ and for $N > 1$ we can write a recurrence relationship that directly mirrors the recursive BST structure: $C_N = N - 1 + (C_0 + C_{N_1})/N + (C_1 + C_{N_2}/N) + ... + (C_{N_1} + C_0)/N$ The N-1 term takes into account that the root contributes 1 to the path length of each of the other N-1 nodes in the tree; the rest of the expression accounts for the subtrees, which are equally likely to be any of the N sizes. After rearranging terms, this recurrence is nearly identical to the one that we solved for quicksort, and we can derive the approximation $C_N \sim 2Nln(N) = 1.39Nlog_2(N)$. Insertions and search misses take one more compare on the average.

5.2.2 Balanced Search Trees

Balanced Search Trees are Binary Search Trees whose performance is guaranteed to be logarithmic, no matter what sequence of keys is used to construct them. In an N-node tree, we would like tthe height to be $\sim log_2(N)$ so that we can guarantee that all searches can be completed in $\sim log_2(N)$ compares, just as for binary search. We will, however, relax the perfect balance requirement a bit to provide guaranteed logarithmic performance for all operations except range search. We will first allow the nodes in our trees to hold more than one key. We will thus now allow 3-nodes (which hold three links and two keys). The middle link are the values between the two keys. A perfectly balanced 2-3 search tree is one whose null links are all the same distance from the root.

Proof 2: Search and insert operations in a 2-3 tree with N keys are guaranteed to visit at most $log_2(N)$ nodes

The height of an N-node 2-3 tree is between $[log_3(N)] = [(log_2(N))/(log_2(3))]$ (if the tree is all 3-nodes) and $[log_2(N)]$ (if the tree is all 2-nodes).

5.2.3 Red-Black Trees

In an RB Tree, we represent the 2-3 tree as a BST. We use the internal left-leaning links as 'glue' for three nodes. These left-leaning links are called the red links. A node can have atmost one red link. Every path from the root to a null link has the same amount of black links (the tree is black-balanced). Every node has one field to determine whether the left link is a red link.
Note: check the powerpoints for insertion/deletion of nodes!

5.2.4 Overview of time complexity

Implementation	Guarantee			Average			ordered iteration?
	search	insert	delete	search hit	insert	delete	
sequential search (linked list)	N	N	N	N/2	N	N/2	no
binary search (ordered array)	$log_2(N)$	N	N	$log_2(N)$	N/2	N/2	yes
BST	N	N	N	$1.39log_2(N)$	$1.39log_2(N)$	?	yes
2-3 tree	$clog_2(N)$	$clog_2(N)$	$clog_2(N)$	$clog_2(N)$	$clog_2(N)$	$clog_2(N)$	yes
red-black BST	$2log_2(N)$	$2log_2(N)$	$2log_2(N)$	$1.00log_2(N)$	$1.00log_2(N)$	$1.00log_2(N)$	yes

5.3 Examenvragen

1. Geef een bespreking van de insert- operaties in een 2-3 boom.
2. Stel dat we ternaire heaps (t.t.z. een heap met 3 kinderen per knoop) zouden gebruiken voor heapsort. Op welke manier zou dit de tijdscomplexiteit beïnvloeden?
3. Vraag 7 examen september 2016 (Young-tabel). Dit is een mooie illustratie hoe heaps niet noodzakelijk in boomstructuren moeten voorgesteld worden, maar bvb. ook een matrix-vorm mogelijk is.
4. Vraag 8, examen juni 2015.

6 Les 8: Greedy algorithms

Greedy algorithms make the choice that seems best *now*. They are used for calculating a route, or compressing data.
Proof 1: no algorithm can compress every bitstring
Proof by contradiction: suppose that you have a universal data compression algorithm U that can compress every bitstring. This would implicate that you can compress bitstring B_0 to a shorter bitstring B_1, which can then in turn be compressed to an even smaller bitstring B_2. Repeat this process until the length is 1. This would mean that every bitstring could be compressed into 1 byte.
Proof by counting: suppose that your algorithm can compress all 1000-bit strings. There are 2^{1000} possible bitstrings with 1000 bits. Only $1 + 2 + 4 + ... + 2^{998} + 2^{999} = 2^{1000} - 1$ bits can be encoded with ≤ 999 bits.
There are several methods to encode data. One of these is Run-Length encoding, in which we use 4-bit counts to represent alternating runs of 0s and 1s. A more widely used encoding system is Huffman encoding. We pick the representation of the characters depending on the specific text. No code (representation) can be a prefix for another code. We construct a prefix tree by greedily linking the two least frequently used characters together. This forms a new character with a frequency equal to the sums of the individual frequencies. Because this implementation requires a PriorityQueue, we get a time complexity of $\sim nlog_2(N)$.
Proof 2: Proof of optimality of Huffman encoding
Lemma 1: any optimal code tree is complete ($\rightarrow$ nodes with one child are impossible)
Lemma 2: there exists an optimal subtree in which the two least frequent letters are siblings at the maximum depth
Every leaf node has a sibling, otherwise the tree would not be complete (lemma 1).
The two least frequent letters x and y must be at the maximum depth. If not, there must be a letter w at maximum depth that occurs more frequently, and that would not be optimal (because w can be switched with x or y).
If x and y are not siblings, swap y with x's sibling. The number of bits for all letters remains unchanged.
We prove the optimality with induction
Base case: only two characters
Assume we can compute the optimal code tree for fewer than r symbols.
T_H is the code tree for set of symbols (s_1, f_1), ..., (s_r, f_r)
$\rightarrow$ s_i and s_j are the first two symbols chosen (lowest frequencies)
$\rightarrow$ s_i and s_j are replaced by symbol (s^*, f_i+f_j)
$\rightarrow$ Algorithm then computes tree with r-1 symbols: this results in tree T^*_H.
$\rightarrow$ W(T_H) = W(T^*_H) + (f_i + f_j)
Consider the most optimal tree T for (s_1, f_1) ... (s_r, f_r)
$\rightarrow$ s_i and s_j must be at the deepest level and siblings (lemma 1 and 2)
$\rightarrow$ Make a new tree T^* by merging s_i and s_j into their parent (s^*, $f_I + f_j$)
$\rightarrow$ W(T) = W(T^*) + (f_i + f_j)
Induction hypothesis: T^*_H is optimal for r-1 letters
$\rightarrow$ $W(T^*_H) <= W(T^*)$
So:
$\rightarrow$ $W(T_H) \quad = W(T^*_H) + (f_i + f_j)$
$\rightarrow$ $\quad <= W(T^*) + (f_i + f_j)$
$\rightarrow$ $\quad <= W(T)$
Hence, T_H is a most optimal tree for r letters.

6.1 Examenvragen

1. Gegeven één of andere string, stel de Huffman coderingsboom op voor deze string. 2. Hoe ziet de Huffman codering er uit indien de frequentie van characters zich verhoudt tot de Fibonnaci getallen? 3. Bewijs dat Huffman-codering optimaal is, t.t.z. de kortst mogelijke gecodeerde string oplevert voor een gegeven te coderen string.

7 Les 9: Minimal Spanning Trees & Shortest Path

7.1 Minimal Spanning Trees

A graph is a set of vertices connected pairwise by edges. In a minimum spanning tree, we try to find the spanning tree (a subgraph T that is connected and acyclic) of an undirected graph G with positive edge weights with the minimum weight. For data representation of a graph, we have a number of options:

→ Maintain a list (linked list or array) of the edges
→ Store the edges in a V-by-V boolean array (arr[v][w] = arr[w][v] = true)
→ Maintain a vertex-indexed array of lists (array where the vertex is the index, each array entry is a list of all the verticies vertex i is connected to)

Proof 1: the cut property

The cut property assumes that the edge weights are distinct and that the graph is connected. It stipulates that given any cut, the crossing edge of minimum weigth is in the MST. A cut in a graph is a partition of its vertices into two nonempty sets and a crossing edge connects a vertex in one set with a vertex in the other.

Proof

Let e be the min-weiht crossing edge in cut
→ Suppose e is not in the MST
→ Consider the graph of adding e to MST
  → This graph has a cycle that contains e
  → Some other edge f in this cycle must be a crossing edge of the cut
→ Removing f and adding e is also a spanning tree
→ Since weight of e is less than the weight of f, that spanning tree has lower weight

Contradiction: e must belong to the MST

7.1.1 Greedy MST algorithm

→ Start with all edges colored gray
→ Find a cut with no black crossing edges
  → color its min-weight edge black
→ Continue until V-1 edges are colored black

7.1.2 Prim's algorithm

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

Correctness of Prim's algorithm: all vertices already in the tree define a cut (→ vertices in the tree vs vertices not in the tree). The minimum connecting edge is part of the MST (this is what Prim's algo does!)

Prim Lazy implementation: a PriorityQueue of edges with at least one endpoint in T. We extract min to determine next edge e=vw to add to T

  → Disregard if both endpoints v and w are in T
  → Otherwise, let v be the vertex not in T: add any edge incident to v to PQ
    and add v to T

Lazy Prim analysis: There are at most E edges in the PQ. Deletion and insertion thus takes about $\sim clog_2(E)$ and the space needed for the PQ is $\sim E$. The time is thus proportional to $Elog_2(E)$ and the space is proportional to E. In practice, however, the PQ is typically much smaller than E. A problem with Lazy Prim, however, is that obsolete edges are kept on the PQ. We will fix this by implementing the following improvements:

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

We now get the following running time for Prim: space $\sim V$ and time $\sim Elog_2(V)$. In practice, for sparse graphs $E \sim V$

7.1.3 Kruskal's algorithm

Kruskal's algorithm is a special case of the greedy MST algorithm. We consider the edges in ascending order of weight, and add the next edge to the tree T unless doing so would create a cycle. Suppose Kruskal's algorithm selects edge e = vw. The cut is the set of vertices connected to v (or w) in tree T. No crossing edge is black (because there are no loops) and no crossing edge has a lower weight. Kruskal runs in $\sim Elog_2(E)$ time. We implement this by maintaining a set of all vertices already in T, and adding loop detection. It is generally slower than Prim's algorithm (there are more edges than vertices).

7.2 Shortest path

7.2.1 Shortest Path Tree

In the shortest path problem, we represent intersections by vertices and roads by edges. The edges have weights and directions. This problem has several variants:

- Source-sink: from one vertex (source) to another (sink)
- Single source: from one vertex to every other
- All pairs: between all pairs of vertices

Edge weights must be non-negative, arbitrary and euclidian. There can be no (negative) cycles. We assume that there exists a shortest path from s to each vertex v. We observe that a shortest path tree (SPT) solution exists. We can represent the SPT with two vertex-indexed arrays:
→ distTo[v] is the length of the shortest path from s to v
→ edgeTo[v] is the last edge on the shortest path from s to v
Edge relaxation is a basic operation. We relax the edge e = v → w. distTo[v] is the length of the shortest known path from s to v, distTo[w] is the length of the shortest known path from s to w and edgeTo[w] is the last edge on the shortest known path from s to w. If e = v → w gives a shorter path to w through v, we update distTo[w] and edgeTo[w]. We present a very generic SPT algorithm:
1. Initialize distTo[s] = 0 abd distTo[v] = ∞ for all v $\neq$ s
2. Repeat until the optimality conditions are satisfied (the shortest path from each v to s has been computed): relax any edge
Optimality conditions: distTo[] are the shortest path distances from s if and only if:
→ for each vertex v, distTo[v] is the length of some path from s to v
→ for each edge e = v → w, distTo[w] $\leq$ distTo[v] + e.weigth()
Proof:
→ Throughout the algorithm, distTo[v] is the length of a simple path from s to v and edgeTo[v] is the last edge on the path
→ Each succesful relaxation decreases distTo[v] for some v
→ distTo[v] can decrease at most a finite number of times

7.2.2 Dijkstra's algorithm

We consider vertices in increasing order of distance from s (non-SPT vertex with the lowest distTo[] value). We then add the vertex to the SPT and relax all edges starting from that vertex.
Proof: Each edge e = v $\rightarrow$ w is relaxed exactly once (when v is added to the SPT), leaving distTo[w] $\leq$ distTo[v] + e.weight(). Inequality holds until the algorithm terminates because distTo[w] cannot increase and distTo[v] will not change (because we have non-negative wieghts and v is not relaxed again). Thus, upon termination, shortest-path optimality conditions hold. This algorithm is almost identical to Prim's algorithm! It also depends on the type of priority queue used. The priority queue contains v vertices and every edge requires a 'decrease key' operation in the queue, thus the total time complexity is $\sim Elog_2(V)$.

7.2.3 Acyclic graphs

For acyclic graphs, a better solutions is possible: we use a topological sorting algorithm. We consider all vertices in topological order, and relax all edges starting at the source vertex. The topological sorting algorithm computes the SPT in any edge-weighted Directed Acylic Graph in $\sim E + V$
Proof: same as Dijkstra

7.2.4 Bellman-Ford

The Bellman-Ford algorithm works, unlike Dijkstra's algorithm, with negative weights. In each pass, it relaxes all edges and it makes V passes. The idea behind it is that after pass i, we've found the shortest path containing at most i edges. It runs in $\sim EV$ time. If distTo[v] does not change during pass i, we do not need to relax any edge starting from v in pass i+1. To implement this, we maintain a queue of vertices whose distTo[] has changed and reuse the values from previous iterations. If after V-1 passes, there are still vertices on the queue, we know that there is a negative cycle.

7.3 Examenvragen

1. Gegeven één of andere grafe, geef de volgorde waarin edges/vertices worden toegevoegd in Kruskal/Prim/Dijkstra/..., of geef de toestand van de prirotiy queue bij Kruskal/Prim/-Dijkstra nadat een specifieke vertex toegevoegd is aan de MST of SPT.
2. Leg uit: algoritme van Prim/Kruskal/DIjkstra. Zou je dit algoritme beschouwen als een greedy algoritme (argumenteer), dan wel als een dynamisch (dynamische algoritmen komen aan bod in les 10) algoritme (argumenteer), beide (argumenteer), of geen van beide (argumenteer).

8 Les 10: String Search

8.1 Brute-force substring search

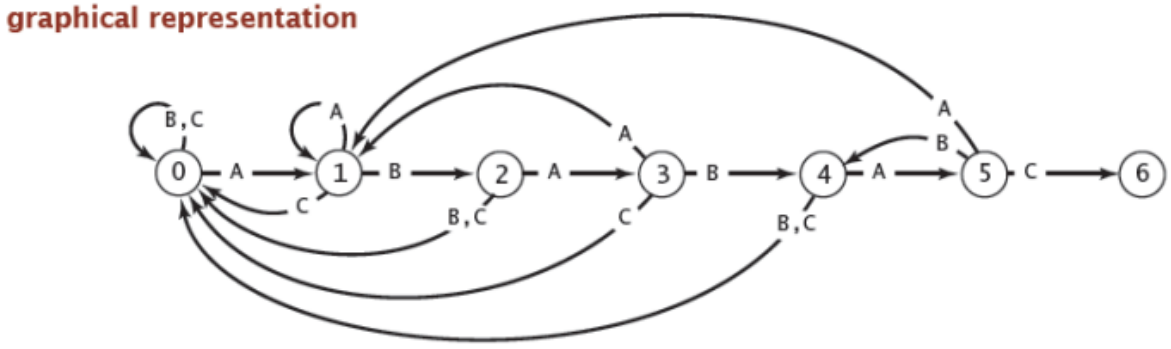
Algoritme 1: Brute-force substring search

```
1  public static int search (String pat, String txt){
2      int M = pat.length;
3      int N = txt.length;
4      for(int i = 0; i <= N - M; i++){
5          int j;
6          for (j = 0; j < M; j++)
7              if (txt.charAt(i + j) != pat.charAt(j))
8                  break;
9          if (j == M) return i; //found
10     }
11     return N;                    //not found
12 }
```

Proof 1: Brute-force substring search requires $\sim NM$ character compares to seach for a pattern of length M in a text of Length N, in the worst case
Proof: A worst-case input is when both pattern and text are all As followed by a B. Then for each of the N - M + 1 possible match positions, all the characters in the pattern are checked against the text, for a total cost of M(N - M + 1). Normally M is very small compared to N, so the total is $\sim NM$.

8.2 Knuth-Morris-Pratt

The basic idea behingd the KMP algortihm is that whenever we detect a mismatch, we already know some of the characters in the text (since they matched the pattern characters prior to the mismatch). We can take advantage of this information to avoid backing up the text pointer over all those known characters.
The Deterministic Finite State Automaton (DFA) The DFA is an abstract string-searching machine with a finite number of states (including start and halt). There is exactly 1 transition for each char in the alphabet. It accepts if a sequence of transitions leads to a halt state.



After reading in *txt[i]*, the state-number is the number of chars that have been matched. The differences from the brute force implementation are that the text pointer i never decrements and that *dfa[][]* needs to be precomputed from the pattern.

Algoritme 2: KMP

```
1  public class KMP{
2      private String pat;
3      private int[][] dfa;
4
5      public KMP(String pat){
6          this.pat = pat;
7          int M = pat.length();
8          int R = 256;
9          dfa = new int[R][M];
10         dfa[pat.charAt(0)][0] = 1;
11         for (int X = 0, j = 1; j < M; j++){
12             for (int c = 0; c < R; c++)
13                 dfa[c][j] = dfa[c][X];
14             dfa[pat.charAt(j)][j] = j+1;
15             X = dfa[pat.charAt(j)][X];
16         }
17     }
18     public int search(String txt){
19         int i, j, N = txt.length(), M = pat.length();
20         for (i = 0, j = 0; i < N && j < M; i++)
21             j = dfa[txt.charAt(i)][j];
22         if (j == M) return i - M;
23         else        return M;
24     }
25 }
```

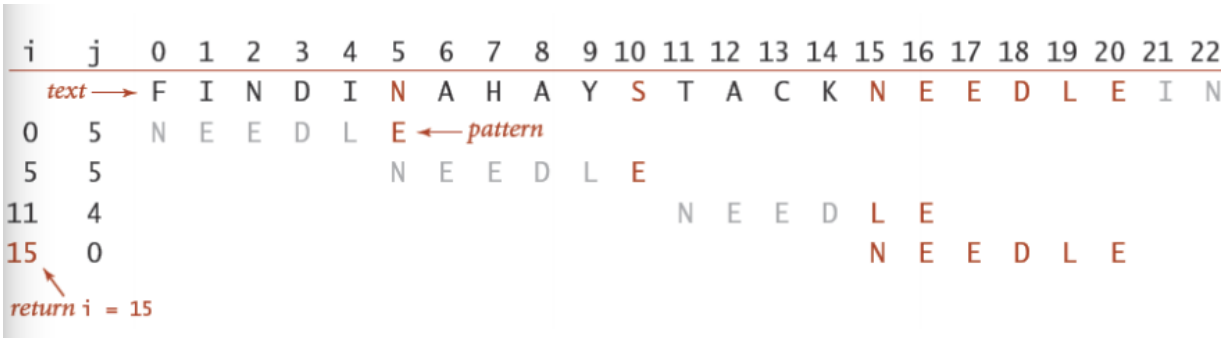
Proof 2: KMP substring search accesses no more that M + N characters to search for a pattern of length M in a text of length N.
Proof: immediate from the code: we access each pattern character once when computing *dfa[][]* and each text character once (in the worst case) in *search()*.
Building the DFA from a pattern:

- Math transition
  If in state j and next char c == pat.charAt(j), then go to state j+1
- Mismatch transition
  If in state j and next char c != pat.charAt(j), then the last j characters of input are pat[1..j-1] followed by c

The KMP constructs *dfa[][]* in time and space proportional to RM, with R the size of the alpabet

8.3 Boyer-Moore

Boyer-Moore scans the characters in the pattern from right to left (you can skip M text characters when finding one not in the pattern).



To implement the mismatched character heuristic, we use an array *right[]* that gives, for each character in the alphabet, the index of the rightmost occurence in the pattern (or -1 if the character is not in the pattern). This value tells us precisely how much to skip if that char appears in the tet and causes a mismatch during the string search.

Algoritme 3: Boyer-Moore

```
1  public class BoyerMoore{
2      private int[] right;
3      private String pat;
4
5      BoyerMoore(String pat){
6          this.pat = pat;
7          int M = pat.length();
8          int R = 256;
9          right = new int[R];
10         for (int c = 0; c < R; c++)
11             right[c] = -1;              //-1 for chars not in pattern
12         for (int j = 0; j < M; j++)    //rightmost position for
13             right[pat.charAt(j)] = j; //chars in the pattern
14     }
15
16     public int search(String txt){
17         int N = txt.length();
18         int M = pat.length();
19         int skip;
20         for (int i = 0; i <= N-M; i += skip){
21             skip = 0;
22             for (int j = 0; i <= N-M; i += skip)
23                 if (pat.charAt(j) != txt.charAt(i + j)){
24                     skip = j - right[txt.charAt(i + j)];
25                     if (skip < 1) skip = 1;
26                     break;
27                 }
28             if (skip == 0) return i;
29         }
30         return N;
31     }
32 }
```

Proof 3: on typical inputs, substring search with the Boyer-Moore mismatched character heuristic uses $\sim N/M$ character compares to search for a pattern of length M in a text of length N
Sketch of the proof: this result can be proved for various random string models, but such models tend to be unrealistic, so we shall skip the details. In many practical situations it is true that all but a few of the alphabet appear nowhere in the pattern, so nearly all ompares lead to M characters being skipped, which gives the stated result.
In the worst case, this becomes MN. With KMP-like rules to avoid repetition, this becomes linear $\sim N$.

8.4 Rabin-Karp

The basic idea behind Rabin-Karp fingerprint search is too compute a hash function for the pattern and then look for a match by using the same hash function for each possible M-character substring. If we find a text substring with the same hash value as the pattern, we can check for a match. A straight-forward implementation would be much slower than bruteforcing, but Rabin and Karp showed that it is easy to compute hash functions for M-character substrings in constant time (after some preprocessing), which leads to a linear-time substring search in practical situations.

$$(a + b)modQ = ((amodQ) + (bmodQ))modQ$$

$$(a * b)modQ = ((amodQ) * (bmodQ))modQ$$

Two variants:

- Las Vegas
  If hash is equal, then check pattern, 100% correct and extremely likely to run in linear time (but worst case MN)

- Monte Carlo
  If hash is equal, we assume pattern is equal as well, always runs in linear time, but probability of 1/Q for mismatch

9 Les 11: Dynamic programming (look-up table)

The basic idea behind dynamic programming is that a problem can be divided into subproblems, and that an optimal solution to subproblems leads to an optimal solution for the problem. Many subproblems share the same subsubproblems etc, and theus we can store the results of the subsubproblems in a look-up table and re-using the results. This can significantly reduce the time-complexity for finding a solution to the problem.

9.1 Examples

9.1.1 Fibonacci

```
1  public static fibonacci (n){
2      int[] fib = new int[n];
3      fib[0] = 0;
4      fib[1] = 1;
5      for (i = 2; i <= n; i++)
6          fib[i] = fib[i-1] + fib[i-2];
7      return fib[n]
8  }
```

9.1.2 Rod cutting

Another common example is rod cutting: start with a rod of integer length n and cut it into several smaller pieces of integer length. What is the best possible cut with the highest value? There are 2^{n-1} possibilities for a rod of length n. A first strategy is cutting the rod in 2 pieces (n-1 possibilities) and recurse on the righmost piece. The same problem (the same rod length) is re-computed often. Another strategy is bottom-up cutting: compute the value of a rod of length 1 and store it in a table. Then, compute a value of a rod of length 2 (you can only cut it into rods of length 1. The value of a rod of length 1 is already computed, so there is no recursion). Then compute the value of longer rods up to length n successively. The optimal values of shorter rods are always computed first so there is no recursion.

9.1.3 Coins in a line

There are an even number of coins, and each coin has a value. Players remove a coin in turns, eiher the leftmost or the rightmost coin. The player that has collected the highest value, wins. We number all coins and store the value of all coins in an array *V[]*. Given a row of coins i to j, player 1 can either gain *V[i]* or *V[j]*. M_{ij} is the maximum value of coins taken by player 1, for coins numbered i to j, assuming player 2 plays optimal. Player 1 must choose based on the following:

- If j = i+1 (only 2 coins left), pick the largest of *V[i]* and *V[j]*.
- Otherwise (more than 2 coins left)
  → If player 1 picks coin i, the gain for player 1 is:
  $\min(M_{i+1,j-1},\ M_{i+2,j})$ + *V[i]*. $\min(M_{i+1,j-1},\ M_{i+2,j})$ is the gain for player 1 after player 2 has picked either j or i+1.
  → If player 1 picks coin j, the gain for player 1 is:
  $\min(M_{i,j-2},\ M_{i+1,j-1})$ + *V[i]*.

For $i = 1, 2, ..., n-1$: $M_{i,i+1} = \max($ *V[i]*, *V[i+1]*$)$
If $j > i+1$: $M_{ij} = \text{maxmin}(M_{i+1,j-1},\ M_{i+2,j})$ + *V[i]*, $\min(M_{i,j-2},\ M_{i+1,j-1})$ + *V[j]*
$j - i + 1$ is always even: initialize table (j-i+1 = 2), compute for all j-i+1 = 4, ... → quadratic time to compute entire table

9.2 Longest Common Subsequence (LCS) problem

A subsequence of string X is any string of the form
X[i_1]X[i_2]X[i_3]X[i_4]X[i_5] with $i_j < i_{j+1}$ for $j = 1, ..., k-1$
Caution! subsequence $\neq$ substring
e.g.: AAAG is a subsequence of CGATAATTGAGA
The LCS problem is that when given 2 strings, what is their longest common subsequence? A naîve solution to this problem is enumerating all possible subsequences of string X (length n) (thus 2^n possibilities) and checking whether it is a subsequence of string Y (lenght m) which can be computed in $\sim m$ time. The total time is then proportional to $2^n m$. We can solve this problem by using dynamic programming: Given strings X (length n) and Y (length m) and the length LCS of X[0 .. i] and Y[0 .. j] = L[i, j]. We must now try to find a way to express L[i, j] as a function of smaller subproblems.
Case 1: suppose X[i] = Y[j] = c. The LCS of X[0..i] and Y[0..j] ends with c.
Proof: Suppose the claim is not true. Then the tlast character of the LCS is different from c. We can then extend the LCS by 1 by adding c. If the last character of the LCS equals c, but the position does not match position i or j in X or Y. Then we can change the position to i or j.
Case 2: suppose X[i] <> Y[j]. The LCS cannot include both X[i] and Y[j]. It can end on X[i], Y[j] or neither but not on both. Hence:

$$L[i,j] = maxL[i-1,j], L[i,j-1] if X[i] \neq Y[j]$$

So:

$$L[i,j] = L[i-1,j-1] + 1 if X[i] = Y[j]$$
$$L[i,j] = maxL[i-1,j], L[i,j-1] if X[i] \neq Y[j]$$

The boundary cases are:

$$L[i,-1] = 0, L[-1,j] = 0$$

The algorithm fills out table L, starting at L[0, 0] and working up. Boundary cases are not explicitly stored in the table. The time complexity is $\sim nm$.

9.3 Optimal Binary Search Trees

Given a sequence $k_1 < k_2 < ... < k_3$ of n sorted keys, each key k_i has search probability p_i. We want to build a binary search tree with minimum expected search cost. The cost for a search is the number of items examined. For key k_i the cost = depth(k_i)+1.
Some observations: the optimal BST may not have the smallest height or the highest-probability key at root. For searches not in the tree, we use dummy nodes d_i with $k_i < d_i < k_{i+1}$. Each dummy value d_i has probabilty q_i of occuring in search. The average search kost can then be determined:

$$cost = \sum_{i=1}^{n}(depth(k_i) + 1) \cdot p + \sum_{i=0}^{n}(depth(d_i) + 1)q_i$$

$$= 1 + \sum_{i=1}^{n} depth(k_i) \cdot p_i + \sum_{i=0}^{n} depth(d_i) \cdot q_i$$

A subtree of a BST contains keys in a contiguous range: $k_i, ..., k_j$ for some $1 \leq i \leq n$ and $d_{i-1}, ..., d_j$. If T is an optimal BST and T contains subree T' with keys $k_i, ..., k_j$ then T' must be an optimal BST for keys $k_i, ..., k_j$. The cost for on optimal search tree on $k_i, ..., k_j$ = c(i, j).
The trivial case: j = i-1 (empty tree); only dummy value d_i; hence c(i, i-1) = q_{i-1}
At least one value k: one of the keys in $k_i, ..., k_j$ (e.g. k_r where $i \leq r \leq j$) must be root of an optimal subtree. The left subtree of k_r contains $k_i, ..., k_{r-1}$. The right subtree of k_r contains $k_{r+1}, ..., k_j$. The optimal left subtree has cost c(i, r-1). The optimal right subtree has cost c(r+1, j). When combined, the depth of each node in the subtrees increases by 1. The cost of the subtree increases by the sum of all probabilities in the subtree.

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

The cost of all probabilities w(i, j) = $\sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$. Also: w(i, j) = w(i, r-1) + p_r + w(r+1, j). Hence:

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

Since we do not know the optimal r, we need to take the minimum over all possible r:

$$c(i, j) = \begin{cases} q_{i-1} \\ minc(i, r-1) + c(r+1, j) + w(i, j) \end{cases}$$

met $i \leq r \leq j$ To implement this, we use the following 2D-arrays:

- c[1..n+1, 0..n] for storing the costs, using only intries in c for which j $\geq$ i-1
- w[1..n+1, 0..n] because we do not want to recompute them every time. w[i,i-1] = q_{i-1} and w[i,j] = w[i,j-1] + p_j + q_j
- root[i,j] stores optimal r for subtree i..j