Die Operationen $Find_P$ und $Union_P$ sind dann folgendermaßen definiert:

$Find_P(v)$: 1. Finde den Knoten v in den Bäumen.
2. Gehe von v aus bis zur Wurzel hoch.
3. Gib den Namen an der Wurzel aus.

$Union_P(i, j)$: 1. Suche Elemente i, j ; i, j sind Bäume mit (i) und (j) als Wurzeln
2. Hänge (i) unter (j)

Die Bäume werden durch ein einziges Array P = P[1..|V|] repräsentiert. Die Einträge in P sind definiert durch:

$$P[v] = \text{Vater von } v$$

Die Operationen sind wie folgt definiert:

$Find_P(v)$: while P[v] ≠ v do v := P[v]
return v

Zeitbedarf: O(|V|) im worst case

$Union_P(i, j)$: P[i] := j

Zeitbedarf: O(1)

11.6. **Algorithmus** (Heuristik Union-by-size)

Prozedur Union-by-size(i, j)

1. if |i| ≤ |j|
2. then 

3. else 

11.7. **Satz** (Baumtiefe mit Union-by-size)

Für alle m ∈ IN und Folgen b = b_1, ..., b_m der Durchführung von Union(i, j) und Find(i), die mit Union-by-size auf die Anfangselemente (1, 1) (2, 1) (3, 1) .... (n, 1) angewendet werden, gilt: Für alle nach b entstandenen Bäume T ist

$$Tiefe(T) \le log_2(|T|) \quad \text{oder} \quad |T| \ge 2^{Tiefe(T)}$$

wobei |T| die Anzahl der Knoten in T ist und Tiefe( ) = 0, Tiefe( ) = 1, Tiefe( ) = 2 usw.

11.8. **Satz** (Zeitbedarf des Kruskal-Algorithmus)

(a) Kruskals Algorithmus mit Union-by-size und ungeordneten Kanten benötigt die Zeit $O(|E|\cdot\log|E|)$.

(b) Kruskals Algorithmus mit Union-by-size und sortierten Kanten benötigt die Zeit $O(|E|\cdot\log|E|)$.

11.9. **Algorithmus** (Wegkompression)

Prozedur Find(v)

1. Lege v auf einen Stack
2. while P[v] ≠ v do
3. v := P[v]
4. Lege v auf den Stack
5. return v
6. for each w auf dem Stack
7. P[w] := v

Die Laufzeit dieser version von Find ist O(Tiefe des Baums, der v enthält), also $O(\log|V|)$ bei Verwendung von Union-by-size und $O(|V|)$ bei Verwendung von Union.

12. Sortieren

12.1. Beispiel

Betrachte den folgenden Sortieralgorithmus, genannt *selection sort*:

Sei A ein Array, der n Zahlen enthält und B ein zweites Array der Länge n. Führe folgende Schritte aus:

1. Suche das größte Element und trage es an die letzte Stelle von B ein.
2. Suche das zweitgrößte Element und trage es an die zweitletzte Stelle von B ein.

.
.
.

Die Laufzeit dieses Algorithmus ist $O(n^2)$, denn für jedes Element müssen maximal n Suchschritte durchgeführt werden.

12.2. **Satz** (Tiefe von Entscheidungsbäumen)

Jeder Entscheidungsbaum T, der n verschiedene Elemente richtig sortiert, hat die Tiefe

$$Tiefe(T) = \Omega(n\cdot\log n)$$

12.3. Beispiel

Ein anderer Sortieralgorithmus.

1. i := 1
2. t := true
3. while i < n – 1 do
4. if A[i] > A[i+1] then t := false ; t = false genau dann, wenn A nicht sortiert
5. i := i + 1

```
6. if A = true  then return A
7.              else sortiere A mit selection sort
```

12.4. **Algorithmus** (Counting Sort)

Prozedur Counting Sort(A, B, k)

```
1. for i = 1 to k              ; die zu sortierenden Elemente sind aus {1, ..., k}
2.    do C[i] := 0
3. for j = 1 to length(A)
4.    do C[A[j]] := C[A[j]] + 1
5. for i = 2 to k
6.    do C[i] := C[i] + C[i–1]
7. for j = length(A) downto 1
8.    do   B[C[A[j]]] := A[j]
9.         C[A[j]] := C[A[j]] – 1
```

Der Zeitbedarf für den Algorithmus Counting Sort wird wie folgt bestimmt: Sei n = length(A).

1. + 2.: O(k)
3. + 4.: O(n)
5. + 6.: O(k)
7. – 9.: O(n)

also insgesamt O(n + k). Ist k = O(n), dann erhält man als Zeitbedarf O(n).

12.5. **Algorithmus** (Radix Sort)

Eingabe: Ein Array A von zu sortierenden Elementen und die Stellenanzahl d.
Ausgabe: Das sortierte Array A.

```
1. for i = 1 to d
2.    do sortiere A auf Stelle i mit einem stabilen Sortieralgorithmus
```

13. Auswahl

13.1. Lemma

Sei A = A[1..n] ein Array von vergleichbaren Elementen. Sei V(n) die Anzahl der Vergleiche um im worst case das Maximum von A zu ermitteln. Dann ist V(n) = n – 1.

13.2. Satz

Sei V(n) die Anzahl der Vergleiche, die nötig sind, um im worst case das Maximum und das Minimum von n Zahlen zu bestimmen. Dann gilt:

$$V(n) = \tfrac{3}{2}n - 2$$

13.3. **Algorithmus** (Simons Select)

Prozedur SimSelect(A, i)

Eingabe: A = A[1..n], i ≤ n.

Ausgabe: Das i-kleinste Element von A.

1. Unterteile A in Gruppen der Größe 5.
2. Bestimme in jeder der $\frac{n}{5}$ Fünfergruppen das mittlere Element (*Median*), z.B. durch Sortieren.
3. if n > 5
4. then Übertrage die Mediane in ein Array B = B[1.. $\frac{n}{5}$].
5. x := SimSelect(B, $\frac{n}{10}$) ; Bestimme den Median der Mediane
6. else x := Median von A
7. Ordne A so um, dass es die Form A‘, x, A“ hat, wobei A‘ = A[1], ..., A[q−1] und A[i] < x für alle i = 1, ..., q−1 und A“ = A[q+1], ..., A[n] und A[j] > x für alle j = q+1, ..., n.
8. if i = q then return x
9. else if i < q then return SimSelect(A‘, i)
10. else return SimSelect(A“, i−q)

13.4. Satz (Laufzeit von SimSelect)

Die Prozedur SimSelect hat die Laufzeit O(n), wobei n die Anzahl der Elemente ist.

14. Divide-and-Conquer I: Multiplikation großer Zahlen

14.1. Algorithmen (Multiplikation in $O(n^2)$)

(a) Schulmethode

$(a_1...a_n)\cdot(b_1...b_n)$

$(a_1...a_n)\cdot b_1$	O(n)
$(a_1...a_n)\cdot b_2$	O(n)
$(a_1...a_n)\cdot b_3$	O(n)
.	.
.	.
.	.
$(a_1...a_n)\cdot b_n$	O(n)
O(n)O(n) ..... O(n)	

} gleiche Konstanten

$n\cdot O(n) = O(n^2)$ für Multiplikationen

$O(n^2)$ für Addition

Die gesamte Laufzeit ist also $O(n^2)$.

(b) Multiplikation mit Divide-and-Conquer.

Sei n eine Zweierpotenz. Zerlege die Zahl $a_1...a_n$ in a‘ = $a_1...a_{\frac{n}{2}}$ und a“ = $a_{\frac{n}{2}+1}...a_n$ und die Zahl $b_1...b_n$ in b‘ = $b_1...b_{\frac{n}{2}}$ und b“ = $b_{\frac{n}{2}+1}...b_n$. Alle vier Teilstücke haben $\frac{n}{2}$ Stellen. Handelt es sich um Zahlen zur Basis 10, dann ist

$$a_1...a_n = a'\cdot 10^{\frac{n}{2}} + a'' \text{ und } b_1...b_n = b'\cdot 10^{\frac{n}{2}} + b''$$

Damit folgt: $a \cdot b = \left(a'\cdot 10^{\frac{n}{2}} + a''\right)\left(b'\cdot 10^{\frac{n}{2}} + b''\right) = a'b'\cdot 10^{n} + a'b''\cdot 10^{\frac{n}{2}} + a''b'\cdot 10^{\frac{n}{2}} + a''b''$.

Mit den Produkten a‘·b‘,a‘·b“,a“·b‘ und a“·b“ fährt man in analoger Weise fort.

Prozedur Mult(X, Y, n)

X und Y sind Zahlen zur Basis 2, $0 \le X, Y < 2^n$, n ist eine Zweierpotenz.

1.a if n = 1then
1.b if X = 1 and Y = 1 then return 1
1.c else return 0
2.a A‘ := Bit 1, ..., Bit $\frac{n}{2}$ von A
2.b A“ := Bit ($\frac{n}{2}$ +1), ..., Bit n von A
3.a B‘ := Bit 1, ..., Bit $\frac{n}{2}$ von B
3.b B“ := Bit ($\frac{n}{2}$ +1), ..., Bit n von B
4. m1 := Mult(A‘, B‘, $\frac{n}{2}$)
5. m2 := Mult(A‘, B“, $\frac{n}{2}$)
6. m3 := Mult(A“, B‘, $\frac{n}{2}$)
7. m4 := Mult(A“, B“, $\frac{n}{2}$)
8. return $\underbrace{\left(m1 \cdot 2^n + m2 \cdot 2^{\frac{n}{2}} + m3 \cdot 2^{\frac{n}{2}} + m4\right)}_{\text{Conquer}}$

Divide

14.2. Satz (Laufzeit der Divide-and-Conquer-Multiplikation)

Die Laufzeit der Prozedur Mult(X, Y, n) ist $O(n^2)$.

14.3. Satz (Laufzeit der Divide-and-Conquer-Multiplikation bei Dreifachunterteilung)

Ist $T(1) = d$
$T(n) = 3 \cdot T(\frac{n}{2}) + d \cdot n$

so ist T(n) von $O(n^{\log 3})$, dies ist von $O(n^{1,59})$.

14.4. Algorithmus (Multiplikation in $O\left(n^{\log 3}\right)$)

Das Aufteilen des Multiplikationsproblems in drei Teile erfolgt in folgender Weise: Zunächst werden die Zahlen a und b wie gehabt in zwei Teile zerlegt. Es ist also

$a = a_1 \ldots a_n$, $a‘ = a_1 \ldots a_{n/2}$, $a“ = a_{n/2+1} \ldots a_n$,
$b = b_1 \ldots b_n$, $b‘ = b_1 \ldots b_{n/2}$, $b“ = b_{n/2+1} \ldots b_n$.

Für Zahlen zur Basis 10 wird folgende Zerlegung vorgenommen:

$$\overbrace{a'b'}^{1.Teilproblem} \cdot 10^n + (\overbrace{(a'-a'')(b''-b')}^{2.Teilproblem} + \overbrace{a'b'}^{1.Teilproblem} + \overbrace{a''b''}^{3.Teilproblem}) \cdot 10^{\frac{n}{2}} + \overbrace{a''b''}^{3.Teilproblem}$$

$$= a'b' \cdot 10^n + (a'b''-a'b'-a''b''+a''b'+a'b'+a''b'') \cdot 10^{\frac{n}{2}} + a''b''$$

$$= a'b' \cdot 10^n + a'b'' \cdot 10^{\frac{n}{2}} + a''b' \cdot 10^{\frac{n}{2}} + a''b'' = ab$$

Programm für Zahlen zur Basis 2:

Prozedur Mult(X, Y, n)

X, Y sind ganze Zahlen mit Vorzeichen, $0 \le |X|, |Y| < 2^n$, n ist eine Zweierpotenz.

1. s := sgn(X)·sgn(Y) ; sgn(X) = +1 – positives Vorzeichen, sgn(X) = -1 – negatives Vorzeichen
2. X := |X|
3. Y := |Y|
4. if n = 1 then
5. if X = 1 and Y = 1
6. then return s
7. else return 0
8. A‘ := Bit 1 ... Bit $\frac{n}{2}$ von A
9. A“ := Bit $\frac{n}{2}$ +1 ... Bit n von A
10. B‘ := Bit 1 ... Bit $\frac{n}{2}$ von B
11. B“ := Bit $\frac{n}{2}$ +1 ... Bit n von B
12. m1 := Mult(A‘, B‘, $\frac{n}{2}$)
13. m2 := Mult(A‘–A“, B‘–B“, $\frac{n}{2}$)
14. m3 := Mult(A“, B“, $\frac{n}{2}$)
15. return(s·(m1·2^n + (m1 + m2 + m3)· $2^{n/2}$ + m3))

Die Prozedur erfüllt die Bedingungen von Satz 14.3, daher ist die Laufzeit $O\left(n^{\log 3}\right)$.

15. Divide-and-conquer II: Matrizenmultiplikation

15.1. Algorithmus (Strassen 1968)

(a) Vorgehensweise wenn n eine Zweierpotenz ist:

1. Setze $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$, $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$
2. Berechne mit $\Theta(n^2)$ skalaren Additionen und Subtraktionen 14 $\frac{n}{2} \times \frac{n}{2}$-Matrizen A_1, B_1, A_2, B_2, ..., A_7, B_7.
3. Berechne rekursiv 7 Matrixprodukte $P_1 = A_1 \cdot B_1$, ..., $P_7 = A_7 \cdot B_7$.
4. Berechne die Matrizen C_{11}, C_{12}, C_{21}, C_{22} mit $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$ aus den P_1, ..., P_7 in $d \cdot n^2$ skalaren Additionen und Subtraktionen.

Dann gilt folgende Rekursionsgleichung:

$$T(1) = d$$
$$T(n) = 7 \cdot T(\tfrac{n}{2}) + d \cdot n^2$$

für geeignetes d. Für T(n) ergibt sich daraus $T(n) = O(n^{\log_2 7})$.

(b) Wahl der Zerlegung der Matrizen A und B.

Die Matrizen A und B sind so in Teilmatrizen A_i und B_i zu zerlegen, dass die Teilmatrizen C_{pq} mit

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

mit 7 Multiplikationen zu berechnen sind. Nach (a) sollen sich die Teilmatrizen C_{11}, C_{12}, C_{21}, C_{22} aus den Teilmatrizen P_1, ..., P_7 ohne zusätzliche Multiplikationen ergeben. Dabei ist $P_i = A_i \cdot B_i$. Es wird gesetzt:

$$A_i = \alpha_{i1} \overbrace{A_{11} + \alpha_{i2} A_{12}}^{\text{obere Hälfte von } A} + \alpha_{i3} \overbrace{A_{21} + \alpha_{i4} A_{22}}^{\text{untere Hälfte von } A}$$

$$B_i = \beta_{i1} \overbrace{B_{11} + \beta_{i2} B_{21}}^{\text{linke Hälfte von } B} + \beta_{i3} \overbrace{B_{12} + \beta_{i4} B_{22}}^{\text{rechte Hälfte von } B}$$

wobei $\alpha_{ij}, \beta_{ij} \in \{-1, 0, +1\}$.

16. Backtracking: Das aussagenlogische Erfüllbarkeitsproblem

16.1. **Definition** (Aussagenlogische Formel)

(a) Sei Var = $\{x_1, x_2, ..., y_1, y_2, ...\}$ eine Menge von Symbolen. Sie werden als *aussagenlogische Variablen* bezeichnet. Die Menge der *aussagenlogischen Formeln* wird induktiv über der Menge der aussagenlogischen Variablen definiert:

1. Ist x eine aussagenlogische Variable, dann ist x eine aussagenlogische Formel.
2. Sind F und G aussagenlogische Formeln, dann sind auch

   $F \wedge G$ (Konjunktion)
   $F \vee G$ (Disjunktion)
   $\neg G$ (Negation)

   aussagenlogische Formeln.

(b) Die Abbildung π ist definiert durch

$$\pi: \text{Var} \to \{0, 1\}$$

Sei F eine aussagenlogische Formel. Die Aussage

$$\pi \models F \quad (\text{F wird wahr unter } \pi, \pi \text{ erfüllt F})$$

ist induktiv über F wie folgt definiert:

$\pi \models x$ für $x \in$ Var $\Leftrightarrow$ $\pi(x) = 1$
$\pi \models F \vee G$ $\Leftrightarrow$ $\pi \models F$ oder $\pi \models G$
$\pi \models F \wedge G$ $\Leftrightarrow$ $\pi \models F$ und $\pi \models G$
$\pi \models \neg F$ $\Leftrightarrow$ nicht $\pi \models F$

(c) Eine Formel F ist erfüllbar genau dann, wenn es ein π gibt mit $\pi \models F$.

16.2. Definition (Konjunktive Normalform)

Eine aussagenlogische Formel F ist in konjunktiver Normalform genau dann, wenn

$$F \equiv \left(L_{11} \vee L_{12} \vee \ldots \vee L_{1k_1}\right) \wedge \left(L_{21} \vee L_{22} \vee \ldots \vee L_{2k_2}\right) \wedge \ldots \wedge \left(L_{m1} \vee L_{m2} \vee \ldots \vee L_{mk_m}\right)$$

Dabei sind die L_{ij} Variable oder negierte Variable (sog. *Literale*).

16.3. Satz (Äquivalenz der konjunktiven Normalform)

Jede Formel F kann in eine Formel G mit $|G| \leq c \cdot |F|$ transformiert werden, so dass gilt:

G ist erfüllbar $\Leftrightarrow$ F ist erfüllbar.

16.4. Lemma (Reduktion von Formeln)

Sei F eine Formel in konjunktiver Normalform und enthalte F die Variable x. $F_{x=1}$ entstehe aus F nach folgender Vorschrift:

1. Entferne jedes Literal L aus F, das x enthält.
2. Streiche $\neg x$ in jedem Literal, in dem es enthalten ist.

$F_{x=0}$ entstehe aus F nach folgender Vorschrift:

1. Entferne jedes Literal L aus F, das $\neg x$ enthält.
2. Streiche x in jedem Literal, in dem es enthalten ist.

Dann gilt: $F_{x=1} \Leftrightarrow F \wedge x = 1$ und $F_{x=0} \Leftrightarrow F \wedge x = 0$

16.5. Algorithmus (Erfüllbarkeit)

Prozedur Davis-Putnam(F)

1. if F = ∅ then return „erfüllbar“
2. if □ in F then return „unerfüllbar“
3. wähle eine Variable x von F
4. H := $F_{x=0}$
5. if Davis-Putnam(H) = „erfüllbar“
6. then return „erfüllbar“
7. H := $F_{x=1}$
8. return Davis-Putnam(H)

Pure-literal-rule: Zeile 3. wird folgendermaßen modifiziert: Kommt in F nur die Variable x vor und nicht $\neg x$, dann ersetze die Zeilen 4. – 7. durch

4. H := $F_{x=1}$
5. return Davis-Putnam(H)

Entsprechend, falls nur die negierte Variable $\neg x$ vorkommt aber nicht x.

Unit-clause-rule: Kommt x ohne Disjunktion vor, d.h. gibt es ein Literal der Form L = x, dann setze x = 1.

Alle bekannten Algorithmen für SAT haben exponentiellen Zeitaufwand!

17. Branch and bound: Das Problem des Handlungsreisenden

17.1. Beispiel

Gegeben sei die Landkarte von Abbildung 17.1, in der Städte mit ihren Koordinaten eingetragen sind.

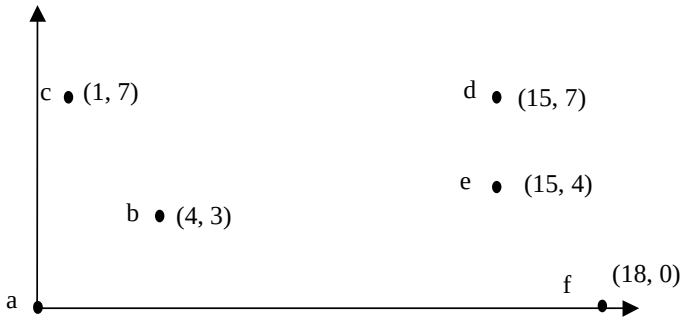

Abbildung 17.1

Als Entfernung zwischen zwei Städten wird der euklidische Abstand genommen. Danach ist

$$\overline{ab} = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

Lösung mit Hilfe des Algorithmus von Kruskal.

Die Orte bilden die Knoten in einem ungerichteten Graphen, die Entfernungen die Gewichte an den Kanten.

Wähle die kürzeste Kante (v, w) so dass die Eigenschaften (1) und (2) erfüllt sind:

(1) Es gibt bisher höchstens eine Kante (v, v‘) und höchstens eine Kante (w, w‘).

(2) Die Knoten v und w liegen bisher in verschiedenen Komponenten, es sei denn, die Kante (v, w) ist die letzte zu bestimmende Kante.

17.2. Beispiel

Gegeben sei der ungerichtete gewichtete Graph von Abbildung 17.2.

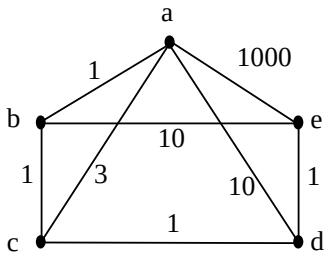

Abbildung 17.2

Mittels der greedy-Strategie erhält man die Rundreise (a, b, c, d, e, a). Ihre Länge ist 1004.

Eine andere Rundreise ist (a, c, b, e, d, a). Bei ihr erfolgt die Auswahl der ersten und der dritten Kante nicht nach der greedy-Strategie. Ihre Länge ist 25.

Bestimmung einer optimalen Rundreise mittels Permutationen:

1. n := |V|
2. für alle n! Permutationen π der Zahlen 1, 2, ..., n do
3. setze $\pi(i) = j \Leftrightarrow$ j ist Nachfolger von i
4. berechne die Kosten der Permutation

Für die Laufzeit dieses Verfahrens gilt $\Omega(n!)$.

Bestimmung einer optimalen Rundreise mittels Backtracking:

Setze bei jeder Kantenauswahl einen Verzweigungspunkt, der die übrigen, noch zur Auswahl in Frage kommenden Kanten enthält.

Bestimmung einer optimalen Rundreise mittels branch and bound:

1. Beginn mit der leeren Rundreise
2. für alle Blattknoten v do
3. expandiere v falls die Bedingungen (1) und (2) erfüllt sind

(1) Kosten(v) = Summe der Kosten der Kanten oberhalb von v ist minimal an allen Nichtblattknoten.

(2) Für jedes bereits so generierte Blatt b des Baums gilt: Kosten(b) > Kosten(k).

17.3. **Algorithmus** (Branch and bound)

1. active := w ; w ist die Wurzel des Baums
2. K[w] := 0 ; Kosten der zu w gehörenden Lösung. Es ist K[w] ≤ Kosten jeder zu w gehörenden Lösung, d.h. K[w] ist untere Schranke.
3. currentbest := ∞ ; Kosten der bisher besten gefundenen Lösung
4. while active ≠ ∅ do
5. k := ein Knoten w aus active mit K[w] minimal
6. active := active – {k}
7. erzeuge die zulässigen Kinder κ_1, ..., κ_k und bestimme K[κ_1], ..., K[κ_k]
; K[κ_i] sind die Kosten der Lösung κ_i, falls κ_i eine Lösung ist
8. for all $\kappa \in \{\kappa_1, ..., \kappa_k\}$ do
9. if K[κ] ≥ currentbest
10. then lösche κ
11. elseif κ ist vollständige Lösung
12. then currentbest := K[κ]
13. else active := active $\cup$ {κ}

Nach dem Durchlauf durch den Algorithmus ist k die Lösung mit K[k] = currentbest.

17.4. **Definition** (Kostenfunktion für Knoten)

Gegeben sei ein gewichteter Graph G = (V, E, d) mit der Kostenfunktion

$$d : V \times V \to \Re^+$$
$$d(v, v) = 0$$

Sei |V| = n und R = (v_1, v_2, ..., v_n, v_1) ein einfacher Kreis. Dann werden die Kosten von R wie folgt bestimmt: