Vortrag zur Seminararbeit mit dem Thema

Divide – and – Conquer

Referent: Janno Rothfos

28 Mai 2008

Grundlagen

Betrachtung der Algorithmen im Detail

- Matrixmultiplikation
- Closest-Pair
- Convex Hull

Zusammenfassung

1. Grundlagen

Die Idee von Divide-and-Conquer

Das Divide-and-Conquer Verfahren beruht auf der Idee, ein Problem solange in Unterprobleme aufzuteilen bis diese elementar gelöst werden können.

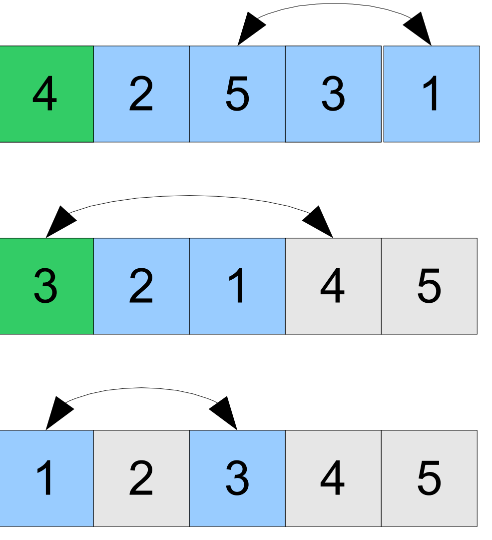
Gemeinsamkeiten von Divide-and-Conquer Algorithmen

- Zerlegung des Gesamtproblems in Teilprobleme
- Lösung der Teilprobleme
- Wenn nötig, Kombination der Teillösungen zu einer Gesamtlösung

Eigenschaften von Divide-and-Conquer Algorithmen

- Laufzeitbestimmung meistens nach: $T(n) = aT(n/b) + f(n) \quad mit \;\; n \in 2^b$
- Typische Laufzeit: $O(n \log(n))$

1. Quicksort:

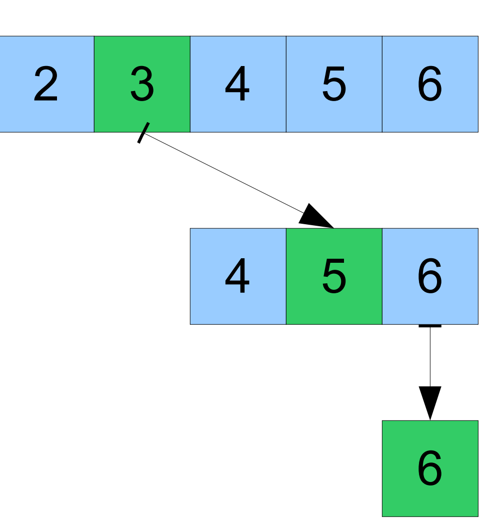


Ansatz:

- Divide: Partitioniere die Menge in abhängig des Pivot-Elementes
- Conquer: Vertausche die Elemente so, dass alle Elemente in der einen Partition kleiner und in der anderen Partition größer als das Pivot-Element sind.

Laufzeit: (Average Case) $O(n \log(n))$

2. Binäresuche



Ansatz:

- Decrease: Suche links wenn das mittlere Element kleiner ist als der Schlüssel, andernfalls suche rechts weiter
- Conquer: Suche das mittlere Element in der Suchmenge

Laufzeit: (Average Case) $O(\log_2(n))$

1. Grundlagen: Das Master-Theorem

Das Master-Theorem dient zur schnellen Laufzeitanalyse für *Divide-and-Conquer*-Algorithmen.

Ausgangspunkt: $T(n) = aT(n/b) + f(n)$

→ Bestimmung von T(n) in Abhängigkeit der Größen a, b und f(n).

Wenn $f(n) \in O(n^d)$, so gilt:

$$T(n) \in \begin{cases} O(n^d) & wenn\ a < b^d \\ O(n^d \log n) & wenn\ a = b^d \\ O(n^{\log_b a}) & wenn\ a > b^d \end{cases}$$

Beispiel: Divide-and-Conquer Ansatz zum Summieren von n Zahlen

Ansatz:

- Divide: Teile die Menge der Summanden in gleiche Teile der Größe n/2
- Conquer: Berechne die Summe für jeden Teil

→ $A(n) = 2A(n/2) + 1$ mit $n \in 2^k$

→ a = 2, b = 2, d = 0 (weil $f(n) \in O(n^0)$) und damit $T(n) \in O(n^{\log_2 2})$ $bzw.\ O(n)$

Bemerkung: Einfache Berechnung der Effizienzklasse, aber Vernachlässigung der multiplikativen Konstanten !

2. Algorithmen im Detail

Strassens Matrixmulitplikation

Nach dem Standard-Algorithmus zur Multiplikation zweier Matrizen A und B wird jedes Element der Ergebnismatrix C nach folgender Formel bestimmt:

$$c_{i_j} = \sum_{k=1}^{m} a_{i_k} b_{k_j}$$

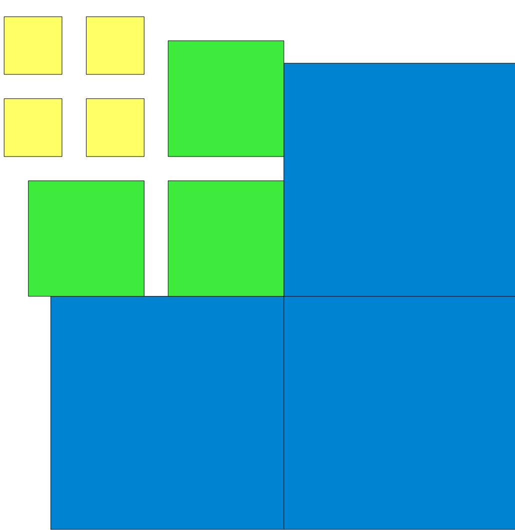
Daraus ergibt sich die Gesamtlaufzeit des Algorithmus für NxN Matrizen:

$$T(n) = n^2 * c_{i_j} = n^2 * n = n^3$$

Und damit die Effizienzklasse:

$$T(n) \in O(n^3)$$

Der Algorithmus von Strassen



1969 zeigte Volker Strassen, das es möglich ist das Produkt C von zwei 2x2 Matrizen A und B mit 7 Multiplikationen und 18 Additionen (bzw. Subtraktionen) durchzuführen.

Der standard Algorithmus braucht im Vergleich 8 Multiplikationen und 4 Additionen.

Vorgehensweise:

Bei dem Algorithmus von Strassen werden die Matrizen A, B und C in jeweils 4 unabhängige Blöcke bzw. Submatrizen eingeteilt.

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

Jeder Ergebnisblock c_i wird in der Folge unabhängig berechnet werden.

Der Algorithmus von Strassen (Fortsetzung)

Dafür werden 7 neue Matrizen m_i Konstruiert.

$$m_1=(a_{00}+a_{11})*(b_{00}+b_{11})$$
$$m_2=(a_{10}+a_{11})*b_{00}$$
$$m_3=a_{00}*(b_{01}-b_{11})$$
$$m_4=a_{11}*(b_{10}-b_{00})$$
$$m_5=(a_{00}+a_{01})*b_{11}$$
$$m_6=(a_{10}-a_{00})*(b_{00}+b_{01})$$
$$m_7=(a_{01}-a_{11})*(b_{10}+b_{11})$$

Anschließend können die Ergebnisblöcke c_i berechnet werden:

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} m_1+m_4-m_5+m_7 & m_3+m_5 \\ m_2+m_4 & m_1+m_3-m_2+m_6 \end{pmatrix}$$

Auch größere Matrizen können gemäß der Blockstruktur in Submatrizen zerlegt und rekursiv gelöst werden.

Einschränkung: Algorithmus ist nur für quadratische Matrizen mit einer Größe von 2^n definiert

➔ Abhilfe: restliche Zeilen mit Nullen auffüllen.

Analyse

Laufzeit: Betrachtung der Multiplikationen

$$M(n) = 7M(n/2) \quad mit \quad n > 1, \; M(1) = 1$$

Effizienzklasse nach dem Master-Theorem: (a=7, b=2, d=0)

→ $M(n) \in O(n^{\log_2 7}) \quad bzw. \quad O(n^{2.807})$

Da die Ersparnis der Multiplikationen nur durch den vermehrten Einsatz von Additionen erkauft wurde, müssen auch die Additionen mit in die Betrachtung einbezogen werden.

$$A(n) = 7A(n/2) + 18(n/2)^2$$

Effizienzklasse nach dem Master-Theorem: (a=7, b=2, d=2)

→ $A(n) \in O(n^{\log_2 7}) \quad bzw. \quad O(n^{2.807})$

Implementierung (Pseudo-Code)

Ein erster Versuch:

```
public function Matrix mult(int size, Matrix m1, Matrix m2) {
   if (size == 1) return new Matrix(1,m1.at(0,0) * m2.at(0,0))
  m1 = mult(size/2, bAdd(00,m1,11,m1,size/2), bAdd(00,m2,11,m2,size/2));
  m2 = ..
  m3 = ..
  m4 = ..
  m5 = ..
  m6 = ..
  m7 = ..
  return combine(m1.add(m4).sub(m5).add(m7), m3.add(m5),...);
}
```

Performance:

- schlechter als die des standard Algorithmus !
- Getestet bis n = 2048

2. Algorithmen im Detail: Strassens Matrixmultiplikation

Ein neuer Versuch:

Motivation: Eliminieren der Combine() Methode.

```
void mult(int *A, int *B, int *R, int n) {
  if (n == 1) {
     (*R) += (*A) * (*B);
  } else {
     mult(A, B, R, n/4);
     mult(A, B+(n/4), R+(n/4), n/4);
     mult(A+2*(n/4), B, R+2*(n/4), n/4);
     mult(A+2*(n/4), B+(n/4), R+3*(n/4), n/4);
     mult(A+(n/4), B+2*(n/4), R, n/4);
     mult(A+(n/4), B+3*(n/4), R+(n/4), n/4);
     mult(A+3*(n/4), B+2*(n/4), R+2*(n/4), n/4);
     mult(A+3*(n/4), B+3*(n/4), R+3*(n/4), n/4);
  }
```

Performance:

8 Multiplikationen... Na das kann der Standard-Algorithmus auch !

Fehlersuche

Für kleine Werte von n kann auch ein Algorithmus mit einer schlechteren Effizienzklasse die besseren Resultate liefern.

Fehleranalyse

Anzahl aller Operationen des Standard-Algorithmus: $M(n) = 2n^3 - n^2$

Anzahl aller Operationen von Strassens Algorithmus: $S(2^k) = 7n^{\log_2 7} - 6n^2$

Ergebnis: Bis n < 654 ! benötigt der Standard-Algorithmus weniger Operationen

Lösung

Wechsel auf den Standard-Algorithmus bei kleinen Werten von n

...aber nicht erst bei n = 654, da Multiplikationen immer noch „teurer“ sind als Additionen.

Die Empfehlungen im Internet schwanken zwischen n = 32 bis n = 128.

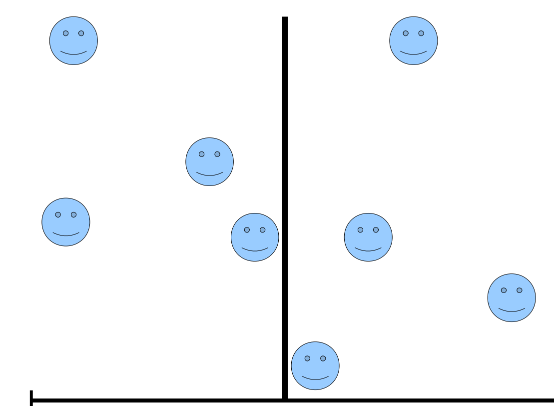
Vorführung des Programms...

Closest-Pair

Problembeschreibung:

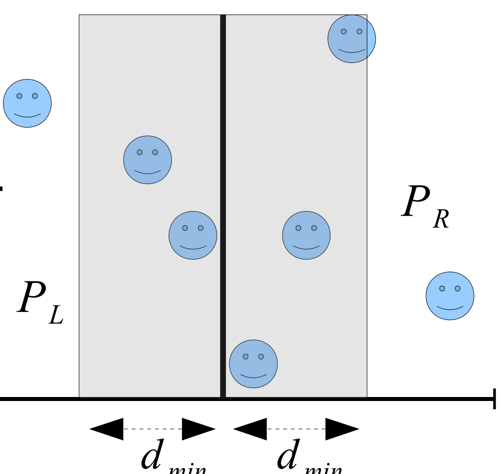
- Gegeben: Eine Menge P von n Punkten $P_i = (x_i, y_i) \quad mit \quad i \in N$
- Gesucht: Die Werte von i und j, die die euklidische Distanz $\sqrt[2]{(x_i - x_j)^2 + (y_i - y_j)^2}$ minimieren
- Bruteforce: Berechne die Distanz zwischen allen $n(n-1)/2$ Paaren und finde das Minimum.
  Der Algorithmus läuft in $O(n^2)$.

Closest-Pair mit Divide-and-Conquer:



- Divide:
  - sortiere die Menge nach X-Koordinaten
  - finde die vertikale Linie die die Menge halbiert
- Conquer:
  - rekursives finden des Closest-Pair in jeder Hälfte
- Combine:
  - überprüfe alle Punkte in der Nähe der Grenze um ggf. das Closest-Pair zwischen zwei Punkten auf unterschiedlichen Seite der Grenze zu finden.

Closest Pair Implementierungs Details:



- Bevor der erste rekursive Aufruf getätigt werden kann muss P sortiert werden.
  - dafür zahlen wir $O(n \log n)$ Rechenzeit
- Danach muss P jedoch kein mal mehr sortiert werden !
- In der sortierten Liste kann die vertikale Trennlinie leicht bestimmt werden.
  - P_L: Soll die Menge aller Punkte auf der linken Seite der Trennlinie sein
  - P_R: Soll die Menge aller Punkte auf der rechten Seite der Trennlinie sein
- Rekursiv:
  - Finde Closest-Pair in P_L und bestimme damit die kürzeste Distanz d_L
  - mache das gleiche für P_R und d_R
  - wichtige Beobachtung: Wenn das Closest-Pair auf unterschiedlichen Seiten der Trennlinie gesucht wird, können nur Punkte mit einem Abstand von $d_{min} = min(d_L, d_R)$ von der Trennlinie mögliche Kandidaten für das Closest-Pair sein.

Closest Pair Implementierungs Details (Fortsetzung)

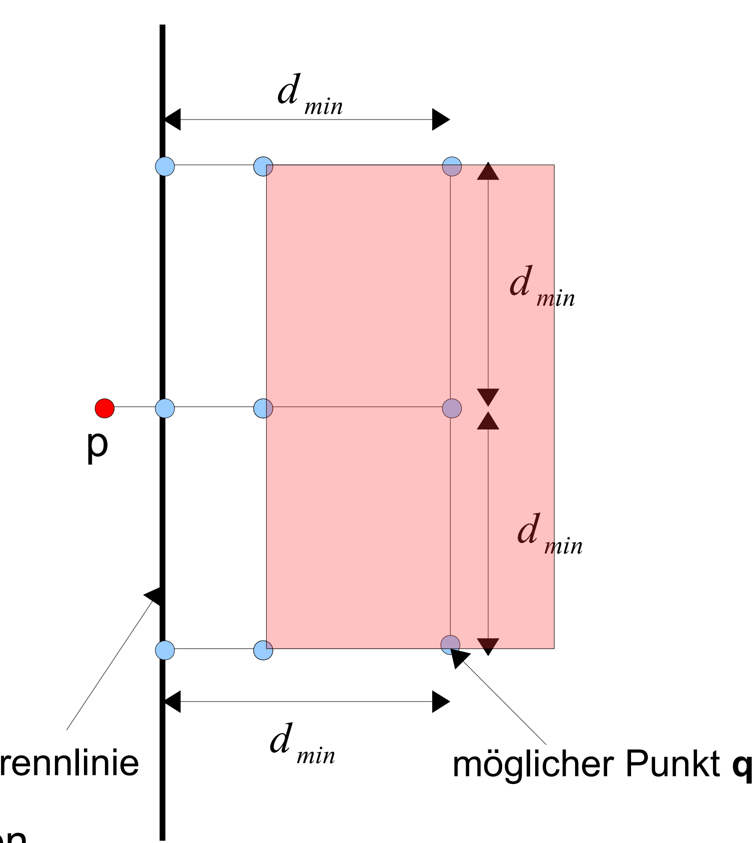
- Noch eine wichtige Beobachtung: Angenommen p und q sind mögliche Kandidaten für das dichteste Paar

  - dann kann q nicht auf der linken Seite liegen

  - außerdem können nur Punkte in P_R mit Y-Koordinaten in einem Intervall von $[y-d_{min}, y+d_{min}]$ erfolgreich mit p gepaart werden.

  - also müssen wir nur die Punkte betrachten die innerhalb einer horizontalen Trennlinie von p in einem Abstand von d_{min} liegen.

  - da alle Punkte einen Mindestabstand von d_{min} zueinander haben, können maximal 6 Punkte als Kandidaten für das Closest-Pair in Frage kommen !



  - um die Suche in der Nähe der Grenze zu beschleunigen ist es sinnvoll die Menge der Kandidaten nach den Y-Koordinaten zu sortieren.

  - da sich der rekursive Abstieg an einer sortieren Liste orientiert, müssen bei dem Aufstieg nur sortierte Listen „gemerged“ werden.
    Für sortierte Listen hat Mergesort eine Effizienzklasse von $O(n)$

Closest Pair Pseudocode

- Divide-Anteil: (grün)
- Conquer-Anteil: (blau)
- Combine-Anteil: (rot)

```
private pset p;

public static pset closestPair(int l, int r) {
  //finde das closest-pair in p[l..r] (sortiert nach x-koordinaten)
  if (p.size() < 2) return infinity;
  mid = (l + r)/2;
  midx = p.get(mid).x; //Trennlinie
  dl = closestPair(l, mid);
  dr = closestPair(mid + 1, r);
  dmin = min(dl, dr);
  cL = selectCandidates(l, mid, dmin, midx);
  cR = selectCandidates(mid + 1, r, dmin, midx);
  merge(l,mid,r); //sortiere p[l..r] nach y-koordinaten
  dm = checkBorder(c, dmin); //closest-pair durch trennlinie getrennt ?
  return min(dm, dl, dr);
}
```

Closest Pair Analyse

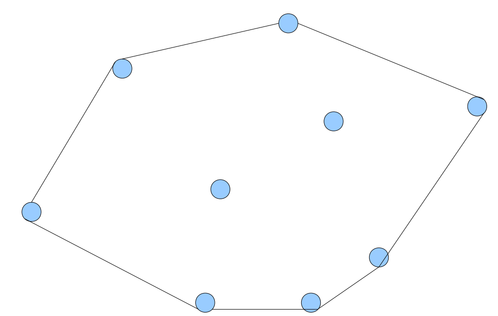
- Die Laufzeit der einzelnen Algorithmen-Teile:
  - Divide: $O(1)$
  - Conquer: $2\mathrm{T}(n/2)$
  - Combine: $O(n)$

- Die Gesamtlaufzeit:
  - Rekurrenzgleichung: $T(n)=2\mathrm{T}(n/2)+O(n)$
  - Laufzeit nach dem Master-Theorem: (a=2, b=2, d=1)

    → $O(n\log n)$

- Besonderheit:
  - Laufzeit für höhere Dimensionen: Ebenfalls $O(n\log n)$
  - Unterliegt damit nicht dem Fluch *curse of Dimensionality*

Convex Hull



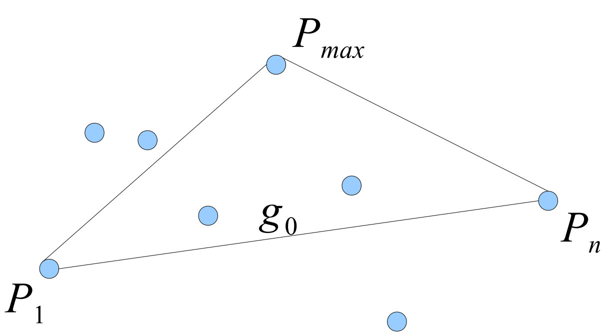
Problembeschreibung:

- Gegeben: Eine Menge P von n Punkten $P_i = (x_i, y_i) \quad mit \quad i \in N$
- Gesucht: Das kleinste konvexe Polygon das alle Punkte von P enthält
- Bruteforce: Berechne alle N^3 Dreiecke und teste für alle n Punkte ob sie innerhalb der berechneten Dreiecke liegen. Dieser Algorithmus läuft in $O(n^4)$

Definitionen:

- konvexes Polygon: Ein Polygon wird als konvex bezeichnet wenn alle Verbindungen zwischen den Punkten eines Polygons innerhalb des Polygons liegen
- Determinante: In der Linearen Algebra ist die Determinante eine spezielle Funktion, die einer quadratischen Matrix eine Zahl zuordnet...

Convex Hull



Convex Hull mit Divide-and-Conquer:

- Vorbereitung:
  - sortiere Menge P nach X-Koordinaten
  - bestimme die ersten beide Punkte P_1, P_n des konvexen Polygons
  - Teile die Menge P in S_1, S_2
  - S_1 enthält alle Punkte links von g_0
  - S_2 enthält alle Punkte rechts von g_0

Nun kann die obere Hülle mit P_1, S_1, P_n und die untere Hülle mit P_1, S_2, P_n rekursiv mit dem eigentlichen Algorithmus bestimmt werden.

- Divide:
  - teile die Menge S_i in eine Menge S_{i1}, die alle Punkte links von $\overrightarrow{p_i p_{max}}$ enthält und in eine Menge S_{i2}, die alle Punkte links von $\overrightarrow{p_{max} p_j}$ enthält.
- Conquer:
  - suche in S_i den Punkt P_{max}, der die größte Entfernung zu g_i hat
  - gibt es zwei Punkte mit identischem Abstand zu g_i, wähle den Punkt, der den Winkel zwischen P_{max}, P_i, P_j maximiert
  - eliminiere alle Punkte die zwischen $\Delta(P_i P_{max} P_j)$ liegen

Convex Hull Implementierungs Details

- P_{max} ist stets ein Punkt der konvexen Hülle
- Die Punkte innerhalb von $\Delta(P_i P_{max} P_j)$ können nicht Bestandteil der konvexen Hüllen sein
- Es gibt daher keine Punkte auf beiden Seiten der Linien $\overrightarrow{p_i p_{max}}$ und $\overrightarrow{p_{max} p_j}$

Implementierung der Geometrischen Funktionen

- Benötigt:
  - eine Funktion die ermittelt, ob ein Punkt links von einer Geraden liegt
  - eine Funktion die den Abstand eines Punktes zu einer Geraden misst
- Abhilfe:
  - für beide Funktionen liefert die folgende Determinante eine Antwort

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1 y_2 + x_3 y_1 + x_2 y_3 - x_3 y_2 - x_2 y_1 - x_1 y_3$$

  - das Ergebnis der Determinante ist positiv, wenn der Punkt $P_3 = (x_{3,} y_3)$ links von der Geraden $\overrightarrow{p_1 p_2}$ liegt.
  - das Ergebnis der Determinante entspricht außerdem zwei mal der Fläche des Dreiecks welches durch $P_{1,} P_{2,} P_3$ aufgespannt wird.
  - P_{max} kann somit ermittelt werden, während die Menge S_i berechnet wird

Convex Hull Analyse

- Laufzeit:
  - Punkt mit größtem Abstand bestimmen und Teilmengen erzeugen erfordert $O(n)$ Zeit
  - Anzahl der Rekursionen abhängig von der Partitionierung der Punkte
  - Best Case: Partitionen sind ungefähr gleich groß

    $\rightarrow\ T(n) = 2T(n/2) + O(n)$ und damit $O(n \log n)$
  - Worst Case: Partitionen sind extrem unausgewogen bzw. klein (Kreis)

    $\rightarrow T(n) = T(n-1) + O(n)$ und damit $O(n^2)$
  - Average Case: bei zufällig verteilten Punkten

    $\rightarrow O(n \log n)$

3. Zusammenfassung

Zusammenfassung

- **Divide-and-Conquer** ist eine Algorithmen-Design-Technik, die Probleme löst in dem sie das Problem in immer kleinere Teilprobleme aufteilt, diese rekursiv löst und wenn nötig, die Teillösungen zu einer Gesamtlösung zusammensetzt.

- **Strassens Matrixmultiplikation** kann relativ einfach mit einem Divide-and-Conquer Ansatz implementiert werden. Die Laufzeit ist $O(N^{2.807})$.
  Es ist jedoch darauf zu achten, das für sehr kleine Matrizen der Standard-Algorithmus verwendet wird.

- **Die Closest-Pair und Convex-Hull** Probleme können beide mit einem Divide-and-Conquer Ansatz gelöst werden. Für beide Algorithmen existieren im Average Case keine besseren Algorithmen.

Vielen Dank für Ihre Aufmerksamkeit