Informatik II: Algorithmen und Datenstrukturen SS 2015

Vorlesung 2b, Mittwoch, 29. April 2015

(Sortieren in Linearzeit, Untere Schranke $n \cdot \log n$)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

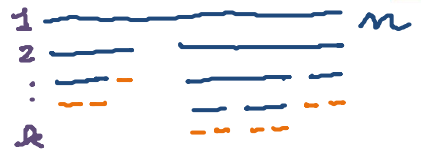
- Laufzeitanalyse Sortieren, Teil 2
  - Quicksort: Laufzeitanalyse im "average case"

    Gestern war: "worst case" (n^2) und "best case" ($n \cdot \log n$)
  - Andere Sortierverfahren: MergeSort, HeapSort

    Kurz die Idee … keine Details und keine Programme
  - Sortieren in Linearzeit: 0-1-Sort und CountingSort

    Das geht aber nur für bestimmte Eingaben
  - Untere Schranke: vergleichsbasiert nicht besser als $n \cdot \log n$
  - **Übungsblatt 2:** Rechenaufgabe zum Logarithmus; Rekursion mit anderer Aufteilung; CountingSort Permutation als Ausgabe

QuickSort im "Average Case" 1/3



- Anzahl Operationen im allgemeinen Fall
  - Sei k die maximale Rekursionstiefe und sei die Größe der Teilfelder auf Stufe j: $n_{j,1}$, $n_{j,2}$, … , n_{j,ℓ_j} (Summe ≤ n)
  - Dann gibt es eine Konstante $A > 0$ so dass gilt:

$$T(n) \le T(n_{1,1}) + T(n_{1,2}) + A \cdot n$$

$$T(n_{1,1}) + T(n_{1,2}) \le T(n_{2,1}) + \ldots + T(n_{2,\ell_1}) + A \cdot n$$

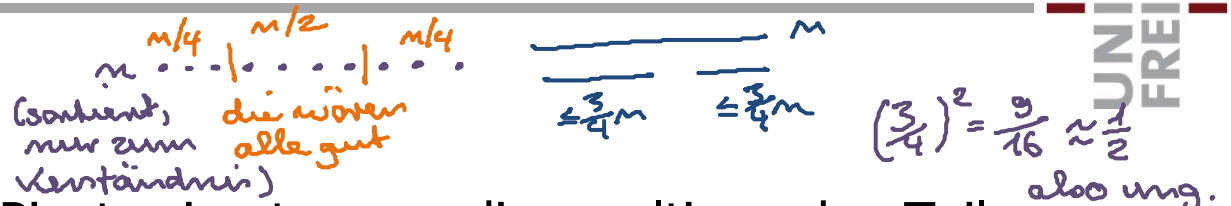
$$T(n_{2,1}) + \ldots + T(n_{2,\ell 2}) \le T(n_{3,1}) + \ldots + T(n_{3,\ell_3}) + A \cdot n \quad \text{usw.}$$

  - Dann kann man leicht zeigen (durch Induktion)

$$T(n) \le T(n_{k,1}) + \ldots + T(n_{k,\ell k}) + k \cdot A \cdot n \le (k+1) \cdot A \cdot n$$

Intuitiv: auf jeder Rekursionsstufe linear viel Arbeit, und insgesamt höchstens k Stufen

QuickSort im "Average Case" 2/3



- Gute Pivots
  - Sagen wir, ein Pivot sei gut, wenn die resultierenden Teilfelder beide $\leq c \cdot m$ groß sind, für $c = \frac{3}{4}$ best case: $c = \frac{1}{2}$

    Dann ist ein zufälliger Pivot gut mit Wahrscheinlichkeit ½
  - Betrachten wir eine Sequenz von ineinander geschachtelten rekursiven Aufrufen: dann ist man nach spätestens $\log_{1/c} n$ guten Pivots bei einem Feld der Größe 1

    $c = \frac{1}{2}: \log_2 n$, $c = \frac{3}{4}: \log_{4/3} n$
  - Daraus folgt: die maximale Rekursionstiefe ist mit hoher Wahrscheinlichkeit $\leq C \cdot \log_{1/c} n$ für eine Konstante $C > 0$

    Das lässt sich mit elementarer Wahrscheinlichkeitsrechnung zeigen, aber lassen wir in dieser (Grund-)Vorlesung weg

QuickSort im "Average Case" 3/3

UNI FREIBURG

- Fazit
  - Auch im "average case" gibt es also eine Konstante $C > 0$, so dass gilt:

    nur das C ist etwas größer als beim „best case"

    $$T(n) \leq C \cdot n \cdot (1 + \log_2 n)$$

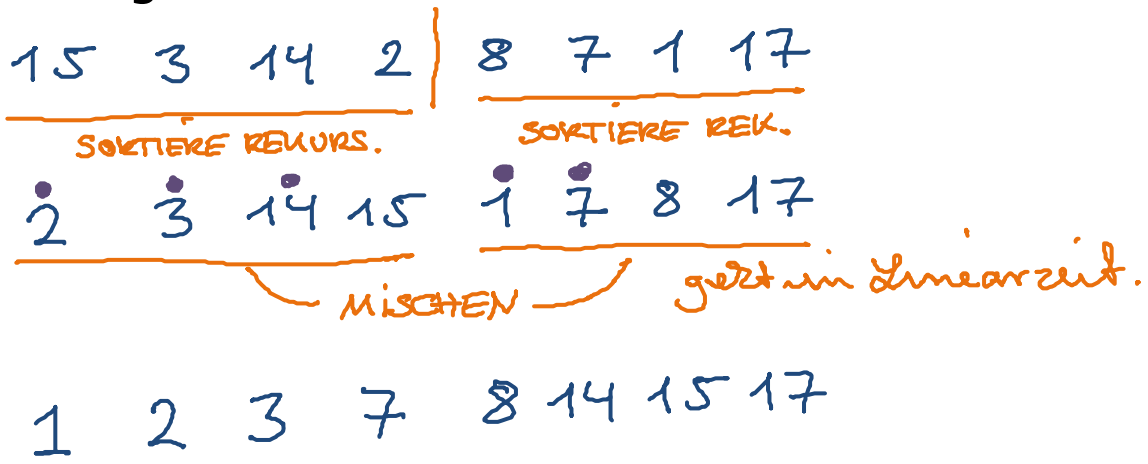
    Quadratische Laufzeit also nur bei ungeschickt gewählten Pivots, und auch dann nur bei bestimmten Eingaben

    Mit dem "Median-of-Three" Pivot bekommt man übrigens auch eine Laufzeit proportional zu $n \cdot \log n$

Andere Sortierverfahren 1/3

UNI FREIBURG

- MergeSort
  - Auch MergeSort teilt das Feld rekursiv auf, aber **immer** in zwei möglichst gleich große Hälften … $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$
  - Es gibt aber keinen "divide" Schritt …dafür müssen die beiden (rekursiv) sortieren Hälften nach den rekursiven Aufrufen noch "gemischt" werden



Andere Sortierverfahren 2/3

UNI FREIBURG

- MergeSort, Fortsetzung
  - Die Laufzeit ist dann **in jedem Fall** wie bei QuickSort im "best case" und im "average case", nämlich

    $T(n) \leq C \cdot n \cdot (1 + \log_2 n)$ für eine Konstante $C > 0$
  - Trotzdem wird man in der Praxis QuickSort bevorzugen:

    Bei geeigneter Wahl des Pivots hat auch QuickSort praktisch immer $T(n) \leq C \cdot n \cdot (1 + \log_2 n)$

    QuickSort macht alles "in place" in dem Eingabefeld

    Für das "Mischen" bei MergeSort braucht man dagegen zusätzlichen Speicher … das kostet Zeit, und führt zu einem höheren Faktor C in der Laufzeit

Andere Sortierverfahren 3/3

UNI FREIBURG

- HeapSort
  - HeapSort benutzt einen **binären Heap** zum Sortieren

    Das ist eine Datenstruktur, die aus einer Menge von n Elementen mit $\leq C \cdot \log n$ Operationen das kleinste Element extrahieren und entfernen kann

    Dazu in einer späteren Vorlesung mehr !

  - HeapSort schafft damit auch **in jedem Fall**

    $T(n) \leq C \cdot n \cdot (1 + \log_2 n)$ für eine Konstante $C > 0$

  - In der Praxis:

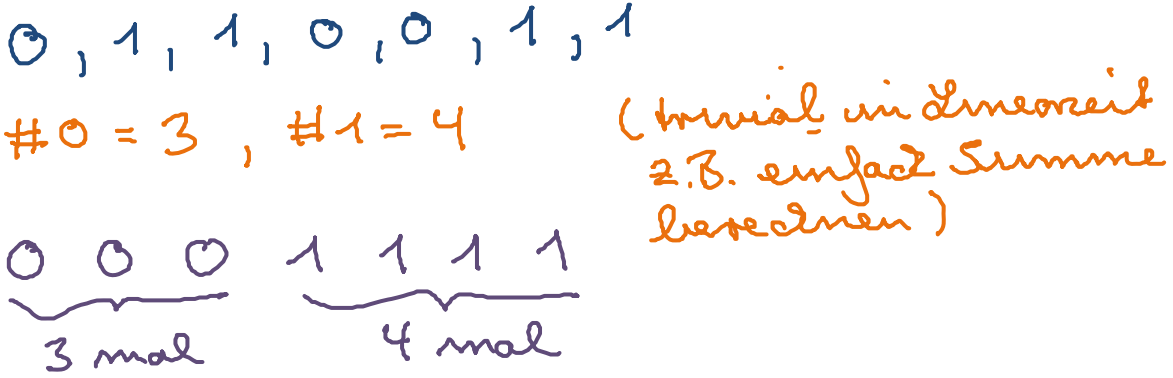
    HeapSort etwas besser als MergeSort … kleineres C
    HeapSort etwas schlechter als QuickSort … größeres C

Sortieren in Linearzeit 1/3

UNI FREIBURG

- ZeroOneSort
  - Geht Sortieren auch schneller als $n \cdot \log n$?
  - Ja, zum Beispiel wenn alle Elemente nur 0 oder 1 sind
  - Dann kann man einfach die 0en und 1en zählen

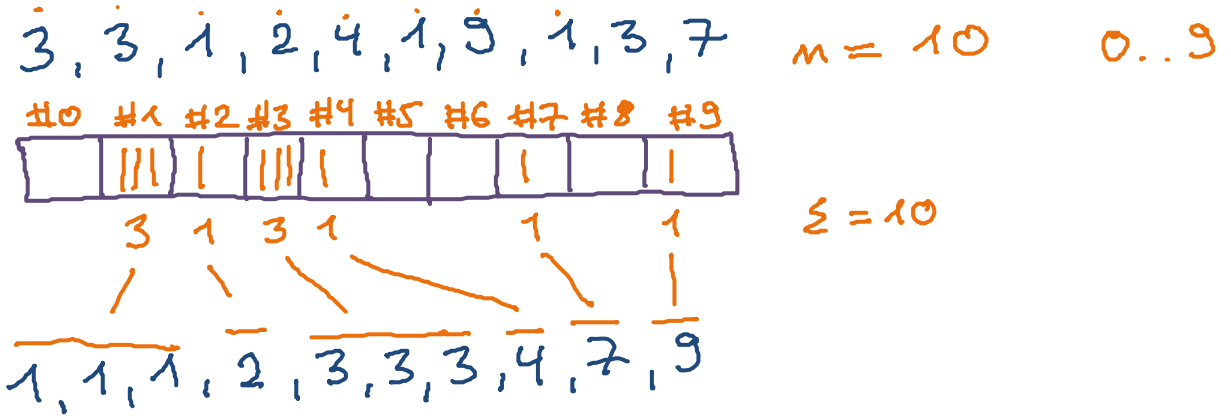
  Dazu schreiben wir gerade ein Programm ZeroOneSort



Sortieren in Linearzeit 2/3

- CountingSort
  - Die Idee klappt auch noch, wenn die Elemente aus dem Bereich 0 .. n – 1 sind

    Dazu schreiben wir gerade ein Programm CountingSort



Sortieren in Linearzeit 3/3

UNI FREIBURG

- Laufzeit
  - Für beide Verfahren gilt offensichtlich

    $T(n) \leq C \cdot n$ ... für ein $C > 0$

    Man geht einmal über das Eingabefeld zum "Zählen", und dann gibt man die (gleich große) Ausgabe aus
  - Ist das vielleicht sogar für beliebige Eingaben möglich?

    CountingSort braucht ein Feld der Größe m für Zahlen aus dem Bereich 0..m-1

    Für m >> n ist die Laufzeit (und der Platzverbrauch) dann proportional zu m, und nicht zu n

Untere Schranke Sortieren 1/12

UNI FREIBURG

- Vergleichsbasiertes Sortieren
  - ZeroOneSort und CountingSort sortieren die Elemente nicht durch "Umsortieren", sondern durch Zählen
  - Wir wollen jetzt zeigen: wenn man "nur Umsortieren" zulässt, geht es tatsächlich nicht schneller als $n \cdot \log n$
  - Dazu müssen wir erst mal genauer fassen, was es heißt, dass ein Algorithmus "nur umsortiert"

Untere Schranke Sortieren 2/12

UNI FREIBURG

- Vorbetrachtung 1
  - Wir werden uns bei unserem Beweis auf Algorithmen **von einer bestimmten Art** beschränken

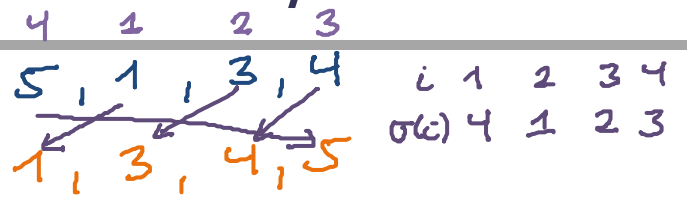
    Wir werden sehen: weil das die Argumentation erleichtert
  - Nehmen wir an, ein Algorithmus **A** ist nicht von dieser Art, aber es gibt einen Algorithmus **A'** von der Art für den gilt:

    Er läuft genauso schnell wie **A**, bis auf $\leq C_1 \cdot n$ Operationen

    Die Ausgabe von **A** kann mit $\leq C_2 \cdot n$ Operationen in die Ausgabe von **A'** überführt werden
  - Dann gilt $T_{A'}(n) \geq n \cdot \log n \quad \Rightarrow \quad T_A(n) \geq n \cdot \log n$

    Wäre **A** schneller, könnten wir auch **A'** schneller machen

Untere Schranke Sortieren 3/12



- Vorbetrachtung 2
  - Bisher haben unsere Algorithmen für Eingabe $x_1, \ldots, x_n$ diese Zahlen in sortierter Reihenfolge ausgegeben
  - Wir wollen im Folgenden Algorithmen betrachten, die stattdessen ausgeben: eine Permutation σ der Zahlen $1, \ldots, n$ so dass $x_{\sigma(1)} \leq x_{\sigma(2)} \leq \ldots \leq x_{\sigma(n)}$
  - **Beobachtung:** alle unseren bisherigen Algorithmen können dahingehend abgewandelt werden, ohne dass sich die Anzahl Operationen um mehr als $C \cdot n$ ändert

    Ü2: das gilt auch für ZeroOneSort und CountingSort

Untere Schranke Sortieren 4/12

UNI FREIBURG

- Vorbetrachtung 3
  - In einem Programm (Python, Java, C++) können an diversen Stellen Verzweigungen auftreten

    while (…) { … }

    for (…) { …}

    if (…) { … } else { … }
  - Ohne Beschränken der Allgemeinheit seien all diese Verzweigungen von der Form if (…) { …} else {… }

    Zum Beispiel ist while (EXPR) { … } äquivalent zu:
    while (true) { if (EXPR) { … } else { break; } }

Untere Schranke Sortieren 5/12

UNI FREIBURG

- Vorbetrachtung 4
  - Betrachten wir die Folge von Entscheidungen in den if (...) { ... } else { ... } Teilen im Ablauf eines Programms
  - Dann entspricht jeder Ablauf einer Folge **IEEIIIEIIIE**...

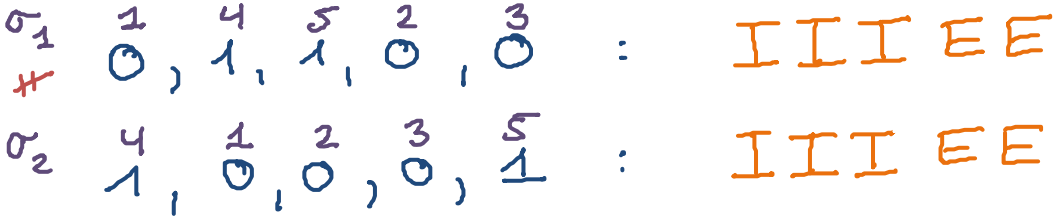
    I = if-Teil wird ausgeführt, **E** = else-Teil wird ausgeführt
  - Ein Algorithmus heißt **vergleichsbasiert**, wenn die I/E Folge die Ergebnispermutation **eindeutig** bestimmt
  - MinSort, QuickSort, MergeSort, HeapSort sind alle vergleichsbasiert bzw. können leicht dazu gemacht werden
  - ZeroOneSort und CountingSort sind es nicht, und können auch nicht dahingehend abgeändert werden

Untere Schranke Sortieren 6/12

UNI FREIBURG

- ZeroOneSort ist nicht "vergleichsbasiert"
  - Hier sind zwei Eingaben mit gleicher I/E Folge aber verschiedenen Ergebnispermutationen

    Wir lassen dabei die I und E von den "for" Schleifen weg, die hängen ja sowieso nicht von der Eingabe ab

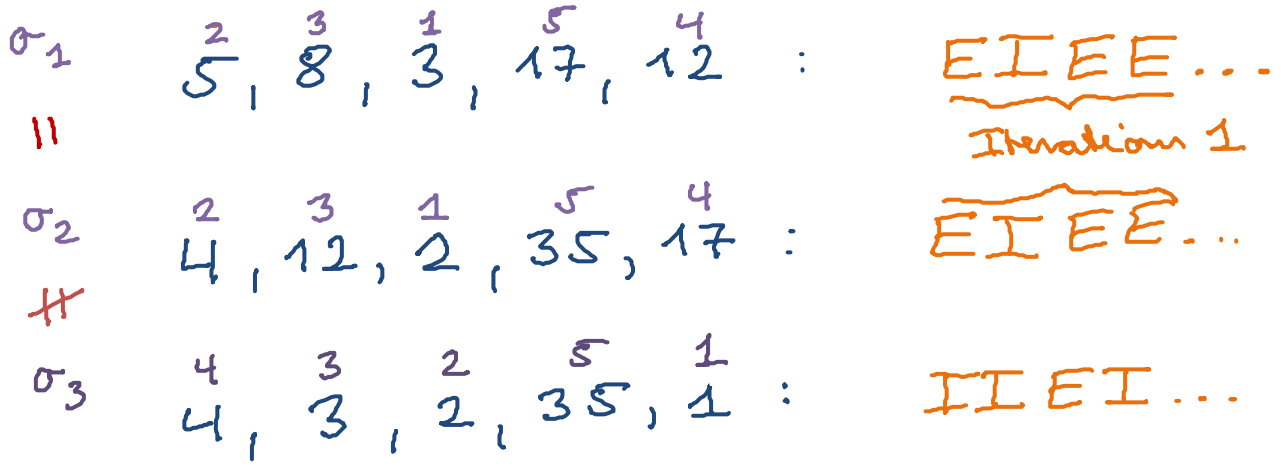


Untere Schranke Sortieren 7/12

UNI FREIBURG

- MinSort ist "vergleichsbasiert"
  - Beispiel: zwei Eingaben mit gleicher I/E Folge und gleicher Permutation, und dritte Eingabe mit anderer I/E Folge und anderer Permutation

Wieder ohne die Is und Es von den "for" Schleifen



Untere Schranke Sortieren 8/12

UNI FREIBURG

- Beweis untere Schranke, Teil 1
  - Wir betrachten jetzt einen beliebigen vergleichsbasierten Algorithmus **A**, der für eine Eingabe der Größe n eine sortierende Permutation σ der Zahlen $1, \ldots, n$ ausgibt
  - Sei $T(n)$ eine obere Schranke für die Anzahl der von **A** benötigten Operationen auf einer Eingabe der Größe n

    Dann höchstens $T(n)$ Verzweigungs-Anweisungen
  - Der Algorithmus gibt also für Eingabegröße n höchstens $2^{T(n)}$ verschiedene Permutationen aus

IIEEIEE...

x viele
$\leq T(n)$

2^x Möglichkeiten
$\leq 2^{T(n)}$

Untere Schranke Sortieren 9/12

UNI FREIBURG

- Beweis untere Schranke, Teil 2
  - Ein korrekter Algorithmus muss für Eingabegröße n alle möglichen Permutationen erzeugen können, das sind $n!$

    Wenn er eine Permutation nicht erzeugen könnte, würde er für die Eingabe, die genau diese Permutation zum Sortieren benötigt, nicht das richtige Ergebnis liefern
  - Auf der vorherigen Folie hatten wir gesehen, dass bei Laufzeit $\leq T(n)$ höchstens $2^{T(n)}$ Permutationen erzeugt werden können
  - Wäre $2^{T(n)} < n!$, würden nicht alle Permutationen erzeugt werden können und der Algorithmus wäre nicht korrekt
  - Es muss also $2^{T(n)} \geq n!$ sein, oder äquivalent $T(n) \geq \log_2(n!)$

Untere Schranke Sortieren 10/12

- Abschätzung von $\log_2(n!)$
  - Dafür wird oft die Stirling-Formel benutzt:

    $n! \geq \sqrt{2 \cdot \pi \cdot n} \cdot (n/e)^n$

  - Wir können das aber auch (etwas weniger genau, aber ausreichend) elementar-mathematisch abschätzen:

(*) $\log_2 x = \frac{\ln x}{\ln 2}$

(**) Stammfunktion von $\ln x$: $x \cdot \ln x - x$

$$\log_2(n!) = \log_2(n \cdot (n-1) \cdot \dots \cdot 1) \qquad \underbrace{\log_2 1}_{=0}$$

$$= \log_2 n + \log_2(n-1) + \dots + \log_2 1$$

$$\overset{(*)}{=} \frac{1}{\ln 2} \sum_{i=1}^{n} \ln i$$

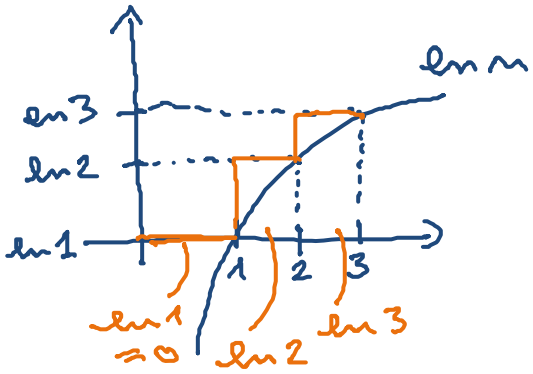
$$\geq \frac{1}{\ln 2} \int_1^n \ln x \, dx$$

$$\geq \frac{1}{\ln 2} \underbrace{x \cdot \ln x - x \Big|_1^n}_{n \cdot \ln n - n + 1 \geq n \cdot (\ln n - 1) \geq n \cdot \frac{\ln n}{2}}$$

$\ln n \geq 2 \Leftrightarrow n \geq e^2$

FAZIT:

$\log(n!) \sim n \cdot \log n$



Untere Schranke Sortieren 11/12

UNI FREIBURG

- Beweis untere Schranke, Zusammenfassung
  - Wir haben gezeigt:

    Sei $T(n)$ eine obere Schranke für einen Algorithmus, der n Elemente vergleichsbasiert sortiert

    Dann ist $T(n) \geq \log_2 (n!) \geq \frac{1}{2} \cdot n \cdot \log_2 n$ für $n \geq 9$

Untere Schranke Sortieren 12/12

UNI FREIBURG

- Fazit
  - Unter den vergleichsbasierten Algorithmen sind also QuickSort, MergeSort, HeapSort alle optimal !
  - Aber in der Praxis zählen auch (unter anderem)

    Konstante Faktoren ... zum Beispiel: "n log n" ist 10 mal schneller als "10 n log n"

    Cache-Effizienz ... Zugriff auf aufeinanderfolgende Elemente im Speicher ist billiger als wenn "verstreut"
  - Zu diesen Aspekten in einer späteren Vorlesung mehr !

Literatur / Links

UNI FREIBURG

- Weiterführende Literatur (bei Interesse)
  - Untere Schranke für vergleichsbasiertes Sortieren

    Mehlhorn/Sanders: 5.3 A Lower Bound [for Sorting]

    Wikipedia: wiki/QuickSort#Formal_analysis