Einführung in die Informatik 2

– Suchen in Datenmengen –

Sven Kosub

AG Algorithmik/Theorie komplexer Systeme
Universität Konstanz

E 202 | Sven.Kosub@uni-konstanz.de | Sprechstunde: nach Vereinbarung

Sommersemester 2010

Lineare Suche

Das elementare Suchproblem:

- Gegeben: abstrakter Datentyp Reihung (Array) F von Elementen
- Methode: Suche Element a in F, d.h. bestimme eine Position $P(F, a)$ eines Elementes a in der Reihung F (-1 falls a nicht in Reihung vorkommt)

Generische lineare Suche:

- „Idee“: Teste jedes Element der Reihung in natürlicher Reihenfolge der Reihung
- Entwurfsschema ist *greedy*

Lineare Suche in Java

Implementierung in Java:

- Reihung f von Elementen des Types `Object`
- Gleichheitstest: (virtuelle) Methode `equals`
- Implementierung von `linearSearch` in Klasse `SearchClass`
- `linearSearch` als `static` deklariert

```
public class SearchClass{
  public static int linearSearch(Object[] f, Object a){
    int i=0;
    while (i<f.length && !(f[i].equals(a))) i++;
    if (i == f.length) return(-1);
    else return(i);
  }
}
```

Lineare Suche mit Wächter

Analyse der Iteration der lineare Suche

```
while (i<f.length && !(f[i].equals(a)))
```

- zwei Tests je Iteration erforderlich
- Test `i<f.length` überflüssig, falls *a* in *f* vorkommt
- führen Wächter (engl. *sentinel*) in *f* ein:
  - füge *a* an das Ende von *f* an
  - nach `while`-Schleife Überprüfung `i==(f.length-1)`: *a* kommt in *f* vor, falls Auswertung `false`, sonst nicht (Wächter gefunden)
  - entferne *a* wieder vom Ende von *f*

Lineare Suche in Java ohne Wächter

- Java führt Zugriffstests auf Arrays automatisch durch
- erzeugt Ausnahme (*exception*) `IndexOutOfBoundsException`, falls `f[i]` ungültigen Index enthält
- statt eines Wächters wird Ausnahme aufgefangen:

```
try{
  while (!f[i].equals(a)) i++;
  return(i);
}
catch (IndexOutOfBoundsException ioobe) {
  return(-1);
}
```

Lineare Suche: Komplexität

Einzelaufwände:

- k_1 Operationen für Initialisierung
- k_2 Operationen für Schleifenabfragen und Rekursion
- k_3 Operationen für Test auf Trivialfall

schlechtester Fall:

- a kommt nicht in f vor
- maximal $n = \texttt{f.length}$ Iterationen, also maximal $(k_1 + k_3) + k_2 n$ Operationen auf allen Eingaben

bester Fall:

- a kommt als erstes Element in f vor
- minimal $k_1 + k_2 + k_3$ Operationen auf allen Eingaben

Lineare Suche: Komplexität

durchschnittlicher Fall:

- wahrscheinlichkeitstheoretische Annahme: nach jedem Element in f wird gleich oft gefragt
- Anzahl der Schleifendurchläufe im Mittel:

$$\sum_{k=1}^{n} \frac{k}{n} = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- im Mittel über alle Eingaben also $(k_1 + k_3) + \frac{k_2}{2}(n+1)$ Operationen

Beurteilung:

- im schlechtesten Fall (*worst case*) und im mittleren Fall (*average case*) lineare Laufzeit

Binäre Suche

divide-and-conquer-Entwurfsschema (Teile und Herrsche):

- zerlege Problem in (kleinere) Teilprobleme
- löse Teilprobleme (i.A. rekursiv)
- füge Lösungen zu Gesamtlösung zusammen

Generische binäre Suche:

- wähle eine Position in f, die f in linke und rechte Teilfolge zerlegt
- (falls Folge sortiert) suche nur in einer der beiden Teilfolgen weiter
- wähle in entsprechender Teilfolge wieder eine Position usw. usf.

Implementierung in Java:

- Reihung f von Elementen von Subtypen von `Comparable`
- Vergleiche: (virtuelle) Methode `compareTo`

Binäre Suche in Java

```
public class SearchClass{
  public static int binarySearch
                    (Comparable[] f, Comparable a, int l, int r){
    int p=(l+r)/2;
    int c=f[p].compareTo(a);
    if (c==0) return(p);
    if (l==r) return(-1);
    if (c<0) {
      if (p>l) return(binarySearch(f,a,l,p-1));
      else return(-1);
    }
    else {
      if (p<r) return(binarySearch(f,a,p+1,r));
      else return(-1);
    }
}
```

Binäre Suche: Komplexität

Einzelaufwände:

- k_1 Operationen für Initialisierung
- k_2 Operationen für Trivialfälle
- k_3 Operationen für Rekursionssteuerung

bester Fall:

- a kommt in f an Position $(l + r)/2$ vor
- minimal $k_1 + k_2 + k_3$ Operationen auf allen Eingaben

schlechtester Fall:

- a kommt in f nicht vor
- maximale Anzahl von Operationen (zur Vereinfachung $n = 2^r$):

$$\begin{aligned} t(n) &= (k_1 + k_2 + k_3) + t(n/2) = 2(k_1 + k_2 + k_3) + t(n/4) = \ldots \\ &= (k_1 + k_2 + k_3) \cdot \log n + (k_1 + k_2) \end{aligned}$$

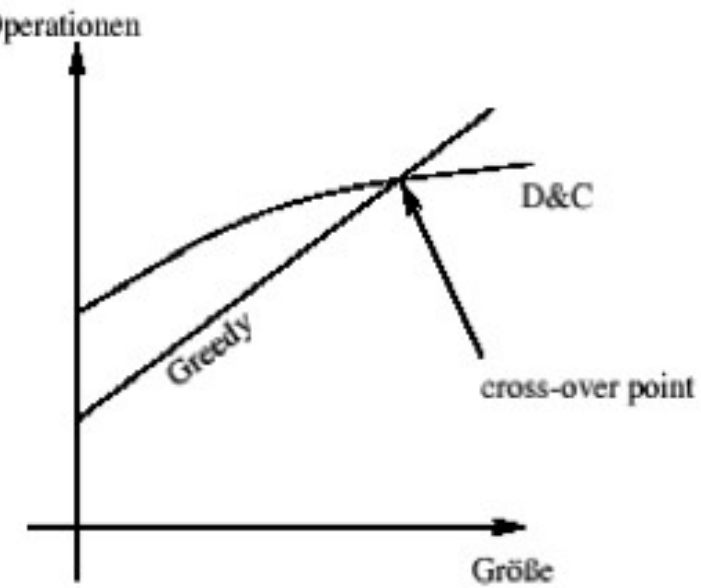
Beurteilung:

- im schlechtesten Fall logarithmische Laufzeit

Cross-over Point

Divide-and-Conquer vs. Greedy:

- Divide-and-Conquer-Verfahren (D&C) mit großem Verwaltungsaufwand (Overhead)
- Greedy-Verfahren mit kleinem Overhead
- im Allgemeinen Übernahmepunkt (Cross-over Point) zwischen beiden Verfahren
- kann experimentell bestimmt werden



Kombinationsverfahren

- benutze Divide-and-Conquer-Verfahren, um große Problem zu behandeln, bis Größe unter Cross-over Point fällt
- nutze Greedy-Verfahren, um kleine Probleme zu lösen
- spezielle, schnelle Implementierungen von Greedy-Verfahren möglich, da nur sehr kleine Probleme behandelt werden müssen
- sehr große Beschleunigungen möglich, da sehr viele kleine Probleme zu lösen sind

Kombinationsverfahren

Frage: Bis zu welcher Problemgröße Greedy-Verfahren einsetzen?

Analyse: Modelliere oder bestimme empirisch Laufzeiten $t_d(n)$, $t_g(n)$ sowie $t_m(n)$ mit Greedy-Einsatz bei $n = 2^m$

Beispielansätze (etwa bei Sortierverfahren):

- $t_d(n) = n + 2t_d(n/2)$ ergibt $t_d(n) = n(\log_2 n + 1)$
- $t_g(n) = kn^2$
- für $k = 0,1$ liegt Cross-over Point zwischen 64 und 128:
  - $n = 32$: $t_g(32) = 0,1 \cdot 32^2 = 102,4 < 192 = 32 \cdot 6 = t_d(32)$
  - $n = 64$: $t_g(64) = 0,1 \cdot 64^2 = 409,6 < 448 = 64 \cdot 7 = t_d(64)$
  - $n = 128$: $t_g(128) = 0,1 \cdot 128^2 = 1638,4 > 1024 = 128 \cdot 8 = t_d(128)$

Kombinationsverfahren

Frage: Bis zu welcher Problemgröße Greedy-Verfahren einsetzen?

Analyse: Modelliere oder bestimme empirisch Laufzeiten $t_d(n)$, $t_g(n)$ sowie $t_m(n)$ mit Greedy-Einsatz bei $n = 2^m$

Beispielansätze (etwa bei Sortierverfahren):

- Modell liefere $t_m(n) = n(\log n - m + k \cdot 2^m)$
- minimiere $t_m(n)$ durch Minimierung von $f(m) = -m + k \cdot 2^m$
- Extremwertbestimmung ($f'(m) = 0$) liefert bei $k = 0,1$ den Wert $m \approx 3,9$
- für $n = 2^{20}$ (=1 Mega), $k = 0,1$ und $m = 4$ ergibt sich:
  $t_d(2^{20}) = 21$ Mops (Mega Operationen)
  $t_g(2^{20}) = 104857,6$ Mops
  $t_4(2^{20}) = 17,6$ Mops

16% Einsparung gegenüber D&C