Bewijzen: Algo & Data

Eigenschappen v. Heaps: (3)

① **Eigenschap 1:** De hoogte v.e heap met n elementen is $h = \lfloor \log_2(n) \rfloor$

- i^{de} niveau heeft 2^i elementen (2^0 … 2^i)
- Diepste niveau aan h heeft tss. de 1 en 2^h nodes

Dit zegt dat een heap met hoogte h minimum $\sum_{i=0}^{h-1} 2^i + 1$ nodes heeft & maximum $\sum_{i=0}^{h} 2^i$ nodes heeft:

$$\sum_{i=0}^{h-1} 2^i + 1 \leq n \leq \sum_{i=0}^{h} 2^i$$ met het feit dat voor een geometrische reeks $\sum_{i=0}^{h} a^i = \frac{a^{h+1} - 1}{a - 1}$, we $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$ krijgen, daardoor is $h \leq \log_2(n) < h+1$ & is dus

$h = \lfloor \log_2(n) \rfloor$

Formule meetkundige rij: $\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$

② **Eigenschap 2:** In een heap met n nodes zijn er $\lceil \frac{n}{2} \rceil$ bladeren

Bewijs via contradictie

- In een heap zitten alle non-leaves in meest linkse locaties v. vector
- En alle leaves zitten in de meest rechtse locaties v.d. vector

leaves $< \lceil \frac{n}{2} \rceil$ 1) Stel dat de heap minder dan $\lceil \frac{n}{2} \rceil$ leaves heeft:

Neem laatste element dat geen leaf is ⇒ element minstens 1 kind op $2i > 2\lceil \frac{n}{2} \rceil \Leftrightarrow 2i > n$

↳ Onmogelijk want totaal # kinderen v.d. heap is n

2) Stel dat de heap meer dan $\lceil \frac{n}{2} \rceil$ leaves heeft:

Neem laatste dat een leaf is ⇒ index leaf = n dus parent is $\frac{n}{2}$ (want ook een leaf moet zijn want # bladeren $> \lceil \frac{n}{2} \rceil$)

↳ Onmogelijk want een leaf heeft geen kinderen

③ **Eigenschap 3:** Er bevinden zich maximum $\lceil \frac{n}{2^{h+1}} \rceil$ knopen op niveau h

- Op hoogte 0 (laag met leaves) ⇒ # bladeren $\lceil \frac{n}{2} \rceil$
- Op hoogte 1 (laag met parent & leaves) ⇒ # $\frac{\lceil \frac{n}{2} \rceil}{2} = \lceil \frac{n}{4} \rceil = \lceil \frac{n}{2^2} \rceil$
- Op hoogte 2 ⇒ $\frac{\lceil \frac{n}{4} \rceil}{2}$ ⇒ $\lceil \frac{n}{8} \rceil$ ⇒ $\lceil \frac{n}{2^3} \rceil$
- Op hoogte h ⇒ $\lceil \frac{n}{2^{h+1}} \rceil$

Performanties v. Sorteeralgoritmen: (3) Simple

① Bubblesort: Performantie

De i'de inner loop doet n-i werk. Dus $b(i) = n-i$

Best case: Innerloop $\bar{w}$ slechts 1x uitgevoerd & has-changed? blijft #f. Outer-loop stop meteen. Dus $r(n) = 1$.

→ Alles is al gesorteerd dus algo gaat geen 2^e x door $O(1)$

$$\sum_{i=1}^{r(n)} b(i) = \sum_{i=1}^{1} n-i = n-1 \in \Omega(n)$$

→ n-1 compares & 0 swaps

Worst case: Innerloop $\bar{w}$ n x uitgevoerd & has-changed? $\bar{w}$ telkens #t. Dus $r(n) = n$

→ grootte el telkens vanvoor (omgekeerd gesorteerd)

$$\sum_{i=1}^{r(n)} b(i) = \sum_{i=1}^{n} n-i = n^2 - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} \in O(n^2)$$

→ $\frac{n(n-1)}{2}$ compares & $\frac{n(n-1)}{2}$ swap (bij elke compare swappen)

② Insertionsort: Performantie

De innerloop zijn werk $\bar{w}$ bepaald door ① value outer-idx orginele ② volgorde v. el in input-vector

Best-case: Orginele input vector al gesorteerd dan $\bar{w}$ de body innerloop nooit uitgevoerd ⇒ $b(n) = 1$ voor uitvoeren v.d test.

⇒ Dus: innerloop 1x ⇒ $\Omega(n)$

Worst-case: Omgekeerd gesorteerde input vector, innerlaag $\bar{w}$ i keer gedaan bij de i^de uitvoering v. outer loop ⇒ $b(i) = i$

⇒ Dus: innerloop n-1 keer met de i'de keer i werk ⇒ $\sum_{i=1}^{n-1} = \frac{n(n-1)}{2}$

⇒ $O(n^2)$

Average-case: Input vector random volgorde dan $\bar{w}$ innerloop $\frac{i}{2}$ x uitgevoerd bij de i^de uitvoering v. outer loop ⇒ $b(i) = \frac{i}{2}$

⇒ Dus: innerloop n-1 x met in de i^de x $\frac{i}{2}$ werk ⇒ $\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4}$

⇒ $O(n^2)$

③ Selectionsort: Performantie

any-case

Outerloop werkt v. 0 naar n-1 & Innerloop werkt v. outer-idx +1 naar n-1. Dus $r(n) = n$ & $b(i) = n-i$ → ALTYD any-case analyse

$$\sum_{i=1}^{r(n)} b(i) = \sum_{i=1}^{n} n-i = n^2 - n = \vartheta(n^2) \rightarrow \text{ALTYD}$$

Slechts n-1 swaps! je vgl tot je kleinste hebt gevonden & dan 1 swap.

④ Quicksort: Performantie

Gearanceerde

Zij $W_i(n)$ de hoeveelheid werk gedaan op recursiediepte i

Best Case: De recursiediepte is $\log_2(n)$ ⇒ pivot valt telkens precies in midden (je vector) → bij complete boom = hoogte

$W_0(n) = n$
$W_1(n) = n-1$
$W_2(n) = n-3$
$W_3(n) = n-7$
↓
alle el's behalve pivot juist te steken

$$W_i(n) = n - (2^i - 1) \Rightarrow \sum_{i=0}^{\log_2(n)} W_i(n) = \sum_{i=0}^{\log_2(n)} n - (2^i - 1)$$

$$(=) \sum_{i=0}^{\log_2(n)} n - \sum_{i=0}^{\log_2(n)} 2^i + \sum_{i=0}^{\log_2(n)} 1$$

$$(=) n \cdot \log_2(n) - \left(\frac{2^{\log_2(n)+1} - 1}{2-1}\right) + \log_2(n)$$

$$(=) n \cdot \log_2(n) - \left(2 \cdot 2^{\log_2(n)} - 1\right) + \log_2(n)$$

$$\in \Omega(n \cdot \log(n)) \quad (=) \quad n \cdot \log_2(n) - (2n - 1) + \log_2(n)$$

Worst Case: De recursiediepte is n ⇒ pivot telkens precies aan een uiteinde.

$W_0(n) = n$
$W_1(n) = n-1$
$W_2(n) = n-2$
$W_3(n) = n-3$

$$W_i(n) = n - i \Rightarrow \sum_{i=0}^{n} (n-i) = n^2 - \sum_{i=0}^{n} i \quad \in O(n^2)$$

⑤ Mergesort: Performantie

diepte level

$W_i(n)$ = de hoeveelheid werk op backtrackniveau i $\quad i \in [1 - \log_2(n)]$

$W_i(n) = 2^{\log_2(n) - i} \; M_i(n)$ → diepste recursie: $\log(n)$
stappen om 2 "regio's" v. lengte 2^{i-1} te mergen naar 1 lengte v. 2.

Dus totale hoeveelheid werk:

$$= \sum_{i=1}^{\log_2 n} W_i(n)$$

→ Bij backtracken geen best/worst case!
⇒ any-case.

~ mergen v. 2 "regio's" ~

worst-case elke element compared

Tussen 2^{i-1} (minste # compares) & $2^i - 1$ (meeste) → Elk element doet 2^{i+1} moves (elke el x2)

Dus # stappen $M_i(n) \leq 2^{i+1} + 2^i - 1$
$\leq 2^{i+2} + 2^i$
$\leq 2^{i+1} + 2^{i+1}$
$\leq 2^{i+2}$

Dus totale hoeveelheid werk:

$$= \sum_{i=1}^{\log_2(n)} W_i(n)$$

$$= \sum_{i=1}^{\log_2(n)} 2^{\log_2(n) - i} \; M_i(n)$$

$$\leq \sum_{i=1}^{\log_2(n)} 2^{\log_2(n) - i} \; 2^{i+2}$$

$$\leq \sum_{i=1}^{\log_2(n)} \frac{2^{\log_2(n)} \; 2^{i+2}}{2^i} = 2 \sum_{i=1}^{\log_2(n)} n$$

$$= O(n \cdot \log_2(n))$$

⑥ Heapsort: performantie

→ $O(n)$. Nadien "extract" n keer delete! wat $O(\log(n))$ is ← uitkalen perform.

→ Dus: $O(n) + O(n \cdot \log(n)) = O(n \cdot \log(n))$
"sterkste slag"

⑦ Heapsort: Stelling

Stelling: geen enkel algoritme dat gebaseerd is op vergelijkingen alleen kan sneller dan $n \cdot \log(n)$

Bew: $n!$ permutaties
- $n!$ bladeren
- minstens $n!$ knopen
- hoogte $h \geq \log(n!)$
- $h \geq n \cdot \log(n) \cdot \frac{1}{n}$
- $\log(n!) \in \Omega(n \cdot \log(n))$

h is de hoogte v.d. beslissingboom
Dus minstens $\Omega(n \cdot \log(n))$ vergelijkingen!

Lemma: $\log(n!) \in \Omega(n \cdot \log(n))$

$\log n!$
$= \log(1) + \log(2) + \ldots + \log(\frac{n}{2}) + \ldots + \log(n)$ (begin, einde)
$\geq \log(\frac{n}{2}) + \ldots + \log(n)$
$\geq \log(\frac{n}{2}) + \ldots + \log(\frac{n}{2})$
$= \frac{n}{2} \log(\frac{n}{2})$
$= \frac{n}{2}(\log(n) - 1)$
$\geq \frac{n}{2} \log(n) - \frac{n}{6} \log(n)$ $n > 8 = 2^3$ $(= n_0)$
Dus $\log(n!) > \frac{1}{3} n \log(n)$ voor $n > 8$

⑧ Radixsort: Performantie + Bucket & counting-sort — Lineair

R: - Radix-iter $\frac{1}{n}$ k keer doorlopen
- In elke iteratie genereren we een "spread" & "collect" v. $O(n)$ werk
→ Dus: $O(k \cdot n)$ (vaak $O(n)$ $\frac{1}{n}$ k constante)

B: $O(n)$ → niet echte sorteeralgo

C: - Door alle inputdata duurt tellen $O(n)$
- Sommen berekenen duurt $O(k)$ (max-key)
- spreiden: $O(n)$

→ alles samen $O(n + k)$ → opnieuw eig $O(n)$

Performantie Heapify: → from "scheme-vector procedure" (1)

1ste poging: sift-down is in $O(\log(n))$ → 2 keer. ⇒ $O(n \cdot \log(n))$

2e poging: Op hoogte h kost sift-down $O(h)$ werk.
Op hoogte h maximaal $\frac{n}{2^{h+1}}$ knopen.
→ Dus totale werk hoogte h: $O(\frac{h \cdot n}{2^{h+1}})$

Dit moet gebeuren voor alle hoogten h tss. 0 & $\log_2 n$

Dus $O(n \cdot \sum_{h=0}^{\log_2(n)} \frac{h}{2^{h+1}})$ ⇒ Dus $O(n)$

$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$

KMP : Performantie

Fase #1: bepalen van σ

evolutie is maatstaf v. vooruitgang

Beschouw $i_p - k$:

- in 1ste tak: van $i_p \geq k$ naar $i_p + 1 \geq k+1$ dus $i_p - k$ ook $\rightarrow \sigma(k) < k$
- in 2e tak: i_p blijft $\geq k$ maar $\sigma(k)$ dus $i_p - k$ verhoogd.
- in 3e tak: i_p naar $i_p + 1 \geq k$ blijft dus is $i_p - k$ verhoogd.

Dus: $i_p - k \leq i_p \geq 0$ dus $i_p \leq n_p$ dus while lus maximaal $2n_p$ keer uitgevoerd.

$\Rightarrow$ Bepalen v. σ is in $O(n_p)$

Fase #2: KMP algoritme zelf

De evolutie v. $i_t + i_p$ in de loop: maatstaf vooruitgang

- match? : $i_t + i_p \Rightarrow i_t + (i_p + 1)$
- geen match?: $i_t + i_p = (i_t + i_p - \sigma(i_p)) + \sigma(i_p) = i_t + i_p$
  Dus ofwel $i_t + i_p$ omhoog, ofwel gaat i_t omhoog.

Som v. $i_t + i_p$ verhoogt / stagneert (maar dan i_t omhoog) in elke stap vd. lus. Dus while max. $2n_t$ keer uitgevoerd.

Totaal

$\Rightarrow$ matchen zelf in in $O(n_t)$

Worst-case: $\text{match}(n_t, n_p) \in O(n_t + n_p)$

Zoeken datawaarde in k-aire zoekboom met n el is $O(\log_2(n))$

- k-aire boom: Boomstructuur waarbij elke node max k kids
- Hoogte boom is $\log_k(n)$ met n el boom met k-arde boom
- Zoeken (doe je op elke niveau) waarbij een deel v.d. zoekruimte geëlimineerd wordt resulteert tot logaritmisch gedrag
- Als basis v. logaritme wij $\log_2(n)$ gebruikt omdat veel algoritmen de neiging hebben om $\log_2$ te gebruiken met basis 2.

$\Rightarrow$ Dus zoeken in een k-aire boom met maximum k-kinderen zal in $O(\log_k(n))$ computationele stappen uitgevoerd worden

uit cursus:

$$\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h}\right)$$

since $\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$ gebruiken

we dit antwoord voor $x = \frac{1}{2}$

wat leidt tot $\sum_{k=0}^{\infty} \frac{k}{2^k} = 2$

we krijgen $\lfloor \log_2(n) \rfloor$

$\Rightarrow O\left(n \sum_{k=0}^{\infty} \frac{k}{2^k}\right) = O\left(n \sum_{k=0}^{\infty} \frac{k}{2^k}\right) =$

$O(n) \rightarrow$ beter resultaat!