Feedback op test

Algemeen

Het is zeer onwaarschijnlijk dat we jullie 2 pagina's code vol laten pennen voor één vraag of functie. Onze oplossing is meestal een tiental regels lang. Maak daar nog 20 regels van als je niet zo sterk bent in het 'opschonen' van code (voorbeelden later), maar geen 40! Dan ben je waarschijnlijk de verkeerde kant op aan het lopen.

Wat te doen als je voelt dat je code te lang wordt?

1. Keer terug naar je schets van de boom/tabel/vector/heap... en leg jezelf opnieuw uit wat jouw methode allemaal zal controleren / verzetten / overlopen om tot de oplossing te komen. Probeer de uitleg zo helder mogelijk te krijgen.
2. Speel dan advocaat van de duivel: kan je methode écht niet falen? Probeer (gericht) een tegenvoorbeeld te vinden. Vind je dat, herdenk je methode dan. Vind je dat niet, dan is je methode waarschijnlijk OK.
3. Probeer ook of je (soms letterlijk) de 'andere kant op kan denken'. Voorbeeld voor de heap: de functie `is_maxheap` moet controleren of de elemten in een tabel aan de heapvoorwaarde voldoen. Of nog: controleer of elke ouder niet kleiner is dan zijn kinderen (zo zijn er 0, 1 of 2). Op tekening wil dit dus zeggen dat je *van boven naar beneden* (van ouder naar kind) controleert.
   Alternatieve formulering: controleer of elk kind niet groter is dan zijn ouder (zo is er maar 1 - uitgenomen voor de wortel). Op tekening wil dit dus zeggen dat je *van beneden naar boven* controleert. Deze denkrichting levert merkelijk minder 'onzekerheden' (aantal kinderen resp. ouders) op, en zal properder te coderen zijn.

Hoe vermijd je lange code?

1. Behandel 'aparte gevallen' (bvb lege lijst, wortel van boom) NIET EERST. Ontwerp je methode / code aan de hand van een zo algemeen mogelijk voorbeeld. Pas als blijkt dat deze code écht verkeerd afloopt voor een speciaal geval, mag je hier aangepaste code voor schrijven. Maar meestal is dat niet nodig.
2. Binnen een for-lus (`for(int i=0; i<n;i++)`) heb je normaal GEEN testen op de teller, toch zeker niet als die test simpel is, zoals `if(i==0)` of `if(i==n)`. Haal dat eerste/laatste geval dan uit de lus. Je code wordt leesbaarder en efficiënter.
3. Is recursie echt nodig / nuttig / tekstbesparend? Als het echt helpt om de 'verdeel en heers'-tactiek toe te passen is recursie OK, maar om bijvoorbeeld na te gaan of alle elementen in een tabel voldoen aan een bepaalde voorwaarde, is recursie niet opportuun. (Voorbeeld: `is_maxheap` uit de test!!)

Veel voorkomende fouten

1. Uiteraard doorloop je NIET de hele tabel als je na enkele stappen al weet wat het antwoord op de uiteindelijke vraag is. (Zodra heapvoorwaarde ergens geschonden wordt, stop je.)

2. Hoe controleer je SNEL of een while-lus een oneindige lus oplevert? Als het item waar je op test NIET opnieuw wordt klaargezet op de laatste regel voor de sluitende accolade, aan de hoofdkantlijn van de lus, is er iets loos. (Niet twijfelen! Dan is er ECHT iets aan de hand!)

3. Let op: `!kleiner(a,b)` is niet hetzelfde als `kleiner(b,a)`, want de negatie van $<$ is $\geq$.

4. Controleer ná het schrijven van je code elke index: al wat tussen [] staat moet binnen de grenzen van de tabel/vector liggen. Voeg niet blindelings extra testen toe om dat te garanderen (maakt je code langer en minder efficiënt), maar vraag je af wat er mis liep aan je methode: waaróm kon je daar buiten de tabel belanden? Voorbeeld:

```
uint teller = 0;
while (resultaat && teller < n / 2) {
    if (kleiner(tabel[teller], tabel[2 * teller + 1])
     || kleiner(tabel[teller], tabel[2 * teller + 2])) {
        resultaat = false;
    }
    teller++;
}
```

   Als grootste waarde voor `teller` heb je `n/2-1`, dus kan `2*teller` gelijk worden aan `n-2`, dus kan `2*teller+2` gelijk worden aan `n`. Gezien de tabel 0-gebaseerd is (zie gebruik van `tabel[teller]` voor `teller=0`), is dit te groot.

Heap: bottom up

Voor wie de methode niet doorhad: volg even deze redenering.
Teken eerst een (would-be-)heap op je blad. NIET in tabelvorm; WEL in boomvorm. De deelbomen waarvan je zeker bent dat ze al voldoen (dus al deelheap zijn), vind je onderaan, op onderste niveau en tweede helft van het voorlaatste niveau: de bladeren. Neem de eerste deelboom die groter is. (Blijkt uit je tekening, en uit redenering: `heap[n/2]` is root van deze deelboom (indexering van heap is 1-gebaseerd).) Je weet dat linkerdeel (1 element) én rechterdeel (0 of 1 element) heap zijn. Schud nu de root op z'n plaats: laat hem zó ver naar beneden zakken tot-ie de heapvoorwaarde niet meer verstoort. Hier zie je dat in de redenering meteen wordt overgegaan naar een meer algemeen geval, dat later in het tijdsverloop opduikt: twee 'grotere' deelbomen die elk apart heap zijn, maar verbonden worden door de gemeenschappelijke ouder van hun wortels. Die ouder is de enige die de heapvoorwaarde kan verstoren, dus wordt hij op z'n plaats geschoven. Mits een gepast voorbeeld (zo eentje van de advocaat van de duivel), zie je dat opschuiven naar beneden niet per se beperkt is tot één niveau. (Opmerking: opschuiven is efficiënter dan wisseloperaties die elkaar opvolgen. Vergelijk met insertion sort.)

Theorie versus praktijk

Veel studenten bepaalden het aantal niveaus dat de (would-be-)heap telt, en voor elk niveau werd dan nagegaan hoeveel elementen er op liggen (gek genoeg werd dan meestal even vergeten dat het laatste niveau niet per se vol is, dus dat ook het voorlaatste niveau niet per se vol niet-bladeren zit!). Nu zijn die formules (met `log` en machten van 2) handig voor de theoretische kant, maar in de praktijk kon het veel eenvoudiger. (Daarvoor dien je eerst een voorbeeld in boomvorm uit te schrijven.) Alle bladeren zitten ná element $n/2$ (indexering 1-gebaseerd), alles wat in $[1, n/2[$ ligt is ouder van 2 kinderen, $n/2$ heeft 1 of 2 kinderen (even wachten voor je hier een speciaal geval van maakt!!). Dus niks geen niveaus nodig (logaritmisch verdeeld), gewoon lineair door de tabel lopen (maar dan wel... van achter naar voor!!).

Hoe kuis je code op?
Hieronder een voorbeeld uit de `maak_heap` opdracht van de test. **Ga niet zoeken naar de al dan niet werkbare delen. Het gaat hier enkel om de vorm.** Dit voorbeeld (zoals er nog vele andere waren...) laat toe om duidelijk aan te geven hoe je - door enkel op de vorm te letten - code kan opkuisen.

```
template <typename T, typename KLEINER>
void maak_maxheap(T *tabel, uint n, const KLEINER &kleiner) {
    for (int i = 0; i < n; i++)
        std::cout << tabel[i] << std::endl;
    int i = n;
    while (i > 1) {
        i = i / 2;
        int indexB = i;
        std::cout << "index : " << indexB << std::endl;
        bool stop = false;
        while (indexB * 2 < n && !stop) {   //+/1
            T temp = tabel[indexB];
            int inK1 = indexB * 2;
            std::cout << "kind1 : " << inK1 << std::endl;
            int inK2 = indexB * 2 + 1;
            std::cout << "kind2 : " << inK2 << std::endl;
            if (inK2 < n - 1) {                    //+/2
                if (kleiner(temp, tabel[inK1])) { //+/3
                    tabel[indexB] = tabel[inK1];
                    indexB = inK1;
                } else if (kleiner(temp, tabel[inK2])) {
                    tabel[indexB] = tabel[inK2];
                    indexB = inK2;
                } else   //+/4
                    stop = true;
            } else
                stop = true;
            //+/5
        }


    }
    for (int i = 0; i < n; i++)
        std::cout << tabel[i] << std::endl;
}
```

1. Eerst en vooral: het eerste item waarop getest wordt in `//+/1` is `indexB`. Dat hoort dus klaargezet te worden op plaats `//+/5` in plaats van verspreid in if/else gedeeltes.

2. Het item `stop` wordt ook niet klaargezet op de juiste plaats. Wat hiervoor nodig is: herdenk waar die aparte boolvariabele voor staat, en noteer deze (waarschijnlijk niet zo moeilijke voorwaarde) onmiddellijk op regel `//+/1` in plaats van via een bool te werken. Dat helpt ons al van het else-gedeelte af (`//+/4`).

3. Het `if/else if`-deel (`//+/2`) bevat tweemaal dezelfde code, op `1` resp. `2` na. Bepaal vooraf of je 1 dan wel 2 nodig zult hebben (bepaal met andere woorden eerst het grootste kind), zodat je ook het `if/else if`-deel (`//+/3`) kwijt bent.

4. De test `if (inK2 < n - 1)` (`//+/2`) is een tussentijdse test in de while-lus die samenhangt met de test van de while-lus zelf nl. `indexB * 2 < n`. Dat is dus tweemaal testen op hetzelfde. Verstreng dan de test in de while-lus: `while(grootstekind .. && ...)`. Dan

moet je uiteraard wel eerst vóór de lus dat grootste kind even klaarzetten. (Dan krijg je een klein stukje duplicated code (item zowel voor als op het einde van de lus klaarzetten) - te vermijden door een functie-aanroep.)

Ultieme tip

Uiteraard is het achteraf altijd makkelijk uitgelegd wat er mis liep op een test, maar daarmee ben je op 't moment zelf niet geholpen. Daarom de belangrijkste tip: neem ruim tijd voor het opstellen van een voorbeeld waarop je de gevraagde methode toepast. Pas als je elke stap hebt kunnen zetten én kan uitleggen waaróm die stap OK is (tegenvoorbeeldje klaarhouden!), kan je de methode verder stroomlijnen (lees: opnieuw uitleggen totdat het er makkelijk uitziet). Dán pas denk je aan code. Hoe makkelijker je uitleg, hoe minder lijntjes code.