Tentamen Algoritmiek
Maandag 10 juni 2024, 13.00 – 16.00 uur

Wanneer er in een opgave gevraagd wordt om uitleg, toelichting of motivatie van je antwoord, is het belangrijk om die ook te geven.
De aantallen punten die bij het begin van elke opgave vermeld worden, zijn indicatief. Ze kunnen dus nog iets wijzigen.
Veel succes!

---

1. [22 pt] Een rijtje van goed-geneste haakjesparen is een string die evenveel openingshaakjes als sluithaakjes bevat, waarbij elk openingshaakje vóór het corresponderende sluithaakje staat. Een voorbeeld van een rijtje van vier goed-geneste haakjesparen is

$$(_1\ (_2\ )_2\ (_3\ )_3\ )_1\ (_4\ )_4 \tag{1}$$

waarbij de subscripts aangeven welk openingshaakje correspondeert met welk sluithaakje. Je kunt dit vergelijken met de accolades in een C++-programma.

   (a) Geef alle rijtjes van drie goed-geneste haakjesparen.

   (b) Stel dat de `string A` een rijtje van goed-geneste haakjesparen is, en stel dat `A[l]` een openingshaakje is. Geef een *niet-recursieve* C++-functie `int sluithaakje (string A, int l)` die de positie r in de string `A` bepaalt (en retourneert) waar het met `A[l]` corresponderende sluithaakje staat.

   Hint: Kijk naar het voorbeeld in (1) om het systeem te zien.

   (c) Je kunt een rijtje van goed-geneste haakjesparen ook als volgt recursief definiëren:

   - Een lege string is een rijtje van (nul) goed-geneste haakjesparen.
   - Een niet-lege string van openingshaakjes en sluithaakjes begint met een openingshaakje. De substring tussen (en exclusief) dit openingshaakje en het corresponderende sluithaakje, en de substring ná het corresponderende sluithaakje zijn allebei ook rijtjes van goed-geneste haakjesparen.
     Bijvoorbeeld, in het rijtje (1) aan het begin van deze opgave zijn ook de substrings $(_2\ )_2\ (_3\ )_3$ en $(_4\ )_4$ rijtjes van goed-geneste haakjesparen.

   Geef, op basis van deze recursieve definitie, een *recursieve* C++-functie `int totaleNesting (string A, int l)` die de zogenaamde *totale nesting* bepaalt van het rijtje goed-geneste haakjesparen dat begint op positie l in string `A`. De totale nesting is het totaal aantal haakjesparen dat binnen andere haakjesparen staat.

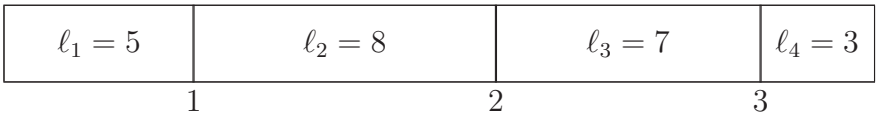
   Bijvoorbeeld: de totale nesting van het rijtje in (1) is 2, omdat de twee haakjesparen $(_2\ )_2$ en $(_3\ )_3$ staan tussen $(_1$ en $)_1$, en er geen haakjes tussen de andere haakjesparen staan. Voor $(_1\ (_2\ (_3\ (_4\ )_4\ )_3\ )_2\ )_1$ is de totale nesting gelijk aan 6, omdat er drie haakjesparen staan tussen $(_1$ en $)_1$, twee haakjesparen tussen $(_2$ en $)_2$, en één haakjespaar tussen $(_3$ en $)_3$. Voor $(_1\ )_1\ (_2\ )_2\ (_3\ )_3\ (_4\ )_4$ is de totale nesting gelijk aan 0.

   Als het goed is, maakt je functie gebruik van het principe van divide-and-conquer. De eerste aanroep van de functie zal van de vorm `totaleNesting (A, 0)` zijn. Je mag er ook bij deze functie vanuitgaan dat `A` daadwerkelijk een rijtje goed-geneste haakjesparen is. Maak desgewenst gebruik van de functie `sluithaakje` uit onderdeel (b).

---

2. [35 pt] Een houtzaagmolen rekent voor het in twee stukken zagen van een stam van lengte ℓ precies ℓ euro, ongeacht de plek waar dit moet gebeuren. Na bestudering van de knoesten op een boomstam van lengte ℓ wordt besloten dat deze in achtereenvolgens (v.l.n.r. gezien) stukken van lengtes $\ell_1, \ell_2, \ldots, \ell_n$ gezaagd moet worden. (De hele boomstam heeft dus lengte $\sum_{i=1}^{n} \ell_i$.) De plekken waar gezaagd gaat worden zijn dus van tevoren bekend. Er zijn $n-1$ zaagplekken. Merk op dat de volgorde van zagen van invloed is op de prijs.

   (a) Stel dat $n = 4$ en $\ell_1 = 5, \ell_2 = 8, \ell_3 = 7, \ell_4 = 3$. De boomstam heeft dus lengte 23.



   Wat zijn de totale kosten als we achtereenvolgens zagen op plekken 1, 2 en 3?
   En wat zijn de totale kosten als we achtereenvolgens zagen op plekken 3, 1 en 2?

   (b) Voor $1 \leq i \leq j \leq n$ definiëren we $L(i, j)$ als de totale lengte van de (deel)stam bestaande uit stukken $i, i+1, \ldots, j$. Ofwel, $L(i, j) = l_i + l_{i+1} + \cdots + l_j$. Als $i < j$ en we dit stuk boomstam in twee stukken willen zagen, kost dat dus $L(i, j)$ euro.

   Geef een *niet-recursieve* C++-functie `void berekenL (int n, int l[])`, die voor elke i en j met $1 \leq i \leq j \leq n$ de lengte $L(i, j)$ berekent en opslaat in een globaal, 2-dimensionaal array `L`, dat gedeclareerd is als `int L[n+1][n+1]`. De array-parameter `l` van deze functie bevat de waardes `l[i]`$= l_i$ voor $i = 1, 2, \ldots, n$.

   Probeer ervoor te zorgen dat de tijdcomplexiteit van je functie kwadratisch is in n. Als dat niet lukt, kun je nog wel het grootste deel van de punten verdienen.

   (c) We willen de totale kosten minimaliseren. Daartoe bekijken we het deelprobleem om de (deel)stam van stukken i t/m j, ter lengte $L(i, j)$ te verzagen tot achtereenvolgens stukken van lengte $\ell_i, \ell_{i+1}, \ldots, \ell_j$. Laat de minimale kosten hiervoor genoteerd worden met $M(i, j)$. Het oorspronkelijke probleem is dan het bepalen van $M(1, n)$.

   Beredeneer dat $M(i, j)$ voldoet aan de volgende recurrente betrekking:

$$M(i,j) = \begin{cases} 0 & \text{als } i = j \\ L(i,j) + \min_{i \leq k \leq j-1}\{M(i,k) + M(k+1,j)\} & \text{als } i < j \end{cases}$$

   (d) Gebruik de recurrente betrekking uit onderdeel (c) voor het bepalen van alle waardes $M(i, j)$ voor het voorbeeld met $n = 4$ uit onderdeel (a).

   (e) Geef een niet-recursieve C++-functie `int bepaalMinKostenBU (int n)` die met behulp van bottom-up dynamisch programmeren, gebruikmakend van de recurrente betrekking uit onderdeel (c), de waarde $M(1, n)$ berekent (en retourneert). Je mag gebruik maken van het globale, 2-dimensionale array `L` dat je bij onderdeel (b) hebt gevuld.

   Vermeld expliciet in welke volgorde de waardes $M(i, j)$ met $1 \leq i \leq j \leq n$ door je functie worden berekend. *Als je geen (complete) functie* `bepaalMinKostenBU` *kunt geven, kun je nog wel enkele punten verdienen door te vermelden in welke volgorde de waardes $M(i, j)$ zouden moeten worden berekend..*

---

3. [34 pt] Laat G_1 een samenhangende, ongerichte graaf zijn met gewichten op de takken.

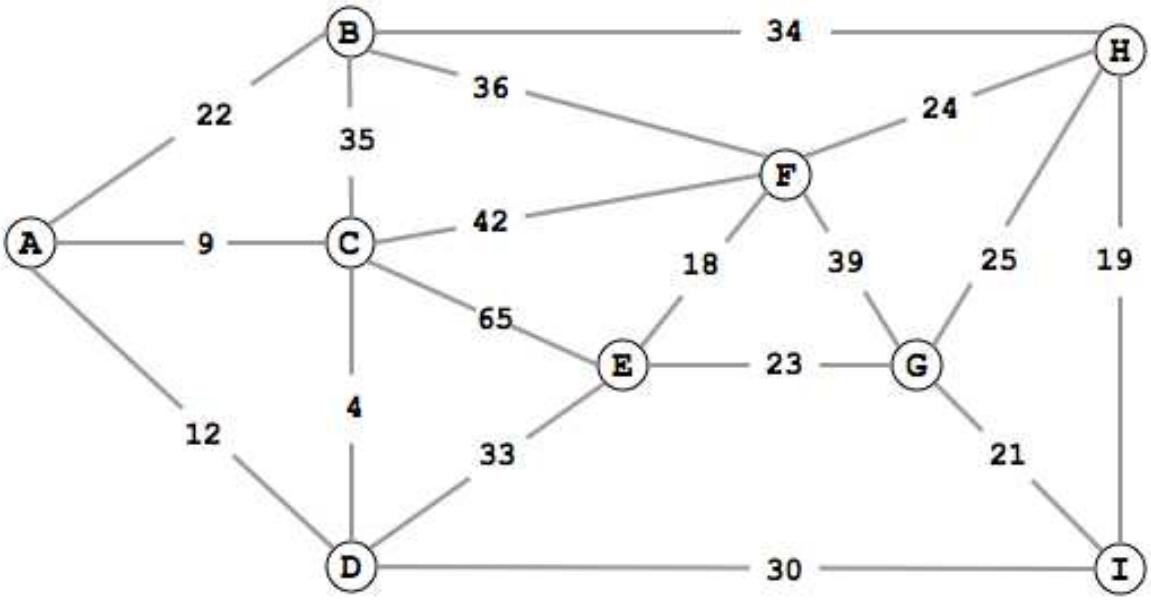
   (a) Wat verstaan we onder een *minimale opspannende boom* van G_1?
   Als je het antwoord op dit onderdeel niet weet, dan kun je het 'kopen' van de docent. Wellicht kun je de hierna volgende onderdelen dan iets gemakkelijker maken.

Het algoritme van Kruskal dient om in een samenhangende, ongerichte graaf met gewichten op de takken een minimale opspannende boom te vinden. Het wordt beschreven door de volgende pseudo-code:

// invoer: samenhangende, ongerichte gewogen graaf $G_1 = (V_1, E_1)$, waarbij $n = |V_1|$ en $m = |E_1|$
// uitvoer: E_T, verzameling takken van minimale opspannende boom T van G_1

sorteer de verzameling takken E_1 op gewicht (in oplopende volgorde): $e_{i_1}, e_{i_2}, \ldots, e_{i_m}$;
$E_T = \emptyset$;
takteller = 0;
k = 0;
while takteller $< n - 1$ **do**
    k = k+1;
    if $E_T \cup \{e_{i_k}\}$ is acyclisch **then**
        $E_T = E_T \cup \{e_{i_k}\}$;
        takteller = takteller+1;
    fi
do

   (b) Laat nu G_1 de volgende graaf zijn:



(Bron: *Stackoverflow*)

Pas het algoritme van Kruskal toe om een minimale opspannende boom van G_1 te bepalen. Geef duidelijk aan, welke takken je in welke volgorde bekijkt en waarom je een tak wel of niet aan je oplossing toevoegt. Teken ook de resulterende boom.

(c) Om te controleren of de graaf met takken uit E_T nog acyclisch is na toevoeging van de tak $e_{i_k} = (a, b)$ (de tak tussen knopen a en b), kun je gebruik maken van depth-first search. Je kunt daarmee namelijk bepalen of knoop b al bereikbaar is vanaf knoop a.

We gaan dit laatste voor een algemene ongerichte graaf $G = (V, E)$ implementeren. Laat $V = \{0, 1, \ldots, n-1\}$, en laat G gerepresenteerd worden met behulp van een adjacency list in `Buur *graaf[n]`, waarbij de klasse Buur als volgt gedefinieerd is:

```
class Buur
{ public:
    int knoopnummer;
    int gewicht;
    Buur* volgende;
};  // Buur
```

Geef een *recursieve* C++-functie `bool bereikbaar (int a, int b, bool bezocht[], Buur *graaf[])` die met behulp van een *depth-first search* controleert of knoop b bereikbaar is vanaf knoop a. Zo ja, dan wordt true geretourneerd, zo nee, dan wordt false geretourneerd. Merk op dat knoop a per definitie bereikbaar is vanaf zichzelf. De parameter `bezocht` bevat bij eerste aanroep van de functie allemaal booleans `false`.

Het is niet de bedoeling dat je bij dit onderdeel gebruik maakt van globale hulpvariabelen. Doe je dat wel, dan verlies je 3 van de punten die je voor dit onderdeel kunt verdienen.

(d) Gegeven is, dat een depth-first search vanuit een enkele knoop in een graaf met n knopen en m takken, bij een adjacency list representatie een tijdcomplexiteit in $\Theta(m)$ heeft.

Stel nu dat we inderdaad gebruik maken van depth-first search om te controleren of de graaf met takken uit E_T nog acyclisch is na toevoeging van de tak $e_{i_k} = (a, b)$, en dat we die graaf inderdaad representeren met een adjacency list. Wat wordt dan de worst-case tijdcomplexiteit (als functie van n en m) van het gehele algoritme van Kruskal, zoals beschreven in de pseudocode aan het begin van deze opgave? Motiveer je antwoord.

---

4. [9 pt] Bij deze opgave hoef je je antwoorden **niet** toe te lichten.

Welke van de drie technieken exhaustive search, backtracking en best-first branch-and-bound

(a) is/zijn alleen geschikt voor optimalisatieproblemen?

(b) bouwt/bouwen oplossingen altijd component voor component op?

(c) houdt/houden bij het component-voor-component opbouwen van oplossingen slechts één deeloplossing tegelijk bij (plus impliciet de daaraan voorafgaande deeloplossingen)?
