Programmeeropdracht 2 — Bergen Beklimmen
Algoritmiek, voorjaar 2026

Inleiding

De vrienden Tadej en Mathieu gaan een fietstochtje door de bergen maken. Als echte liefhebber wil Tadej zo veel mogelijk bergen beklimmen. Mathieu bouwt het liefst een beetje geleidelijk op: eerst een lage berg, en vervolgens steeds hogere bergen. Daarnaast moeten ze een beetje op de centen letten. Vanwege de populariteit van het bergfietsen wordt er bij iedere berg een bepaalde prijs gerekend, die je moet betalen als je de berg (met z'n tweeën) wil opfietsen. Ze hebben maar een beperkt budget. Kun jij ze helpen om bij een gegeven serie bergen zoveel mogelijk bergen te kiezen die ze met deze randvoorwaarden kunnen beklimmen, en dan nog zoveel mogelijk geld over te houden?

Specificatie

Onze vrienden kunnen kiezen uit n bergen, met $1 \leq n \leq 500$. De bergen zijn genummerd van 1 t/m n. Elke berg i heeft een prijs p_i en een hoogte h_i met $1 \leq p_i \leq 1.000.000$ en $h_i \geq 1$. Het beschikbare budget is $b \geq 1$. De bergen die ze op de fiets gaan beklimmen, moeten in volgorde van nummering beklommen worden. Stel dus dat ze bergen 2, 5 en 10 beklimmen, dan moet dat in die volgorde, en berg 5 moet (strict) hoger zijn dan berg 2 en berg 10 moet (strict) hoger zijn dan berg 5.

Het doel is dus om een maximaal aantal van k bergen te bepalen, die in de juiste volgorde binnen het budget b beklommen kunnen worden. Als er meerdere verzamelingen van k bergen zijn die hieraan voldoen, kies dan een verzameling met een zo laag mogelijke totaalprijs.

Alle getallen die het probleem specificeren (n, b, de prijzen en de hoogtes) zijn integers.

Een verzameling bergen in het probleem kan bijvoorbeeld als volgt beschreven worden:

```
5
3 100
2 200
5 150
4 300
1 400
```

We hebben hier $n = 5$ bergen. De eerste berg heeft prijs $p_1 = 3$ en hoogte $h_1 = 100$, enzovoort. Met een budget $b = 10$ kunnen Tadej en Mathieu maximaal $k = 4$ bergen beklimmen, namelijk bergen 1, 2, 4 en 5, die een totaalprijs van precies 10 kosten. Met een buget $b = 9$ kunnen ze maximaal $k = 3$ bergen beklimmen, en de goedkoopste drie bergen die ze kunnen beklimmen zijn 1, 2 en 5, met totaalprijs 6.

Eisen aan het programma

Je moet een C++-programma schrijven dat **met behulp van dynamisch programmeren** het probleem voor Tadej en Mathieu oplost. Dat kun je doen met een twee-dimensionaal array `minKosten`, waarbij `minKosten`$[i][k]$ de minimale kosten voorstelt om (precies) k van de bergen

$1, 2, \ldots, i$ te beklimmen, **inclusief berg i zelf**. Als het niet mogelijk is om k van deze bergen te beklimmen, moet je een geschikte default-waarde voor `minKosten`$[i][k]$ gebruiken.

Voor het voorbeeld hierboven met $n = 5$ bergen en budget $b = 9$ ziet het array `minKosten` er als volgt uit:

	$k =$				
	1	2	3	4	5
$i = 1$	3	-	-	-	-
2	2	5	-	-	-
3	5	8	-	-	-
4	4	6	9	-	-
5	1	3	6	-	-

Een - in de tabel betekent dat de combinatie niet mogelijk is. Uiteindelijk gaat het er dan om, om de hoogste waarde van k te vinden waarbij er een i bestaat waarvoor `minKosten`$[i][k]$ niet de defaultwaarde heeft. Vervolgens zoeken we bij die waarde k de waarde i waarvoor `minKosten`$[i][k]$ minimaal is. In het voorbeeld wordt dat dus $k = 3$ en $i = 5$.

Het is aan te raden om, voordat je begint met programmeren, goed na te denken over de recurrente betrekking waaraan `minKosten`$[i][k]$ voldoet. In het verslag wordt hier ook naar gevraagd.

Het kan overigens gebeuren dat de vrienden geen enkele berg kunnen beklimmen, omdat alle prijzen hoger zijn dan hun budget. Natuurlijk willen Tadej en Mathieu niet alleen weten wat de minimale kosten zijn om zo veel mogelijk bergen te beklimmen, maar ook welke bergen er dan beklommen moeten worden. Je programma moet die informatie ook opleveren. In principe kunnen er overigens meerdere oplossingen zijn die dezelfde minimale kosten geven. Het maakt dan niet uit welke oplossing je programma oplevert.

Een complete specificatie van het probleem met de hand invoeren wordt veel werk. Daarom moet je programma een tekstbestand in kunnen lezen, dat de vorm heeft als in het eerder gegeven voorbeeld. Hoewel er direct boven dat voorbeeld voorwaarden worden beschreven waaraan de getallen voldoen, moet je programma die voorwaarden nog wel controleren. De gebruiker voert vervolgens handmatig het budget b in.

Voor u beschikbaar

Op de website van het vak

`https://liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/`

is een pakketje beschikbaar, met daarin

- een skeletprogramma,
- een tekstbestand `gebergte1.txt` met het voorbeeld uit deze specificatie,

In het skeletprogramma wordt een klasse `Gebergte` gedefinieerd, die door het hoofdprogramma gebruikt wordt. Het programma wordt onder Linux gecompileerd met het commando `make`. Vervolgens kun je het met het commando `./KlimReisPlanner` runnen. Je krijgt dan een menu aangeboden, met de keuze om een instantie in te lezen, random te genereren of een experiment te doen. Bij de eerste twee keuzes kun je vervolgens uit vier mogelijkheden kiezen om het probleem voor die instantie (en een gekozen budget) op te lossen:

1. rechtstreeks recursief,
2. met top-down dynamisch programmeren,
3. met bottom-up dynamisch programmeren,
4. met een gretig algoritme.

Bedoeling is om de `TODO`'s in `gebergte.h`, `gebergte.cc` en `main.cc` uit te voeren. Je mag hierbij gebruik maken van standaardfuncties in `standaard.h/cc`, zoals de functie `randomGetal` die een random getal tussen (en inclusief) twee grenzen oplevert.

De meeste functies die je moet implementeren worden toegelicht in het skeletprogramma, speciaal in `gebergte.h`. Goed om te weten: als er in het commentaar boven een functie `Pre:` staat, dan volgt de **preconditie** voor die functie: een aantal aannames waar je vanuit mag gaan als je in die functie binnenkomt. Je hoeft dat dus in de functie niet meer te controleren. De gebruiker van de functie moet daar van tevoren voor zorgen. Net zo staat `Post:` voor de **postconditie** van de functie: zaken die na afloop van de functie moeten gelden. Daar moet je in de functie dus voor zorgen.
Van een aantal functies geven we nu een nadere toelichting:

- De belangrijkste functies zijn `losOpRec`, `losOpTD` en `losOpBU`. Alle drie deze functies bepalen het maximaal aantal bergen k en de bijbehorende minimale kosten voor de huidige instantie en een budget b. Hoewel ze dat op verschillende manieren doen, moeten ze wel allemaal gebaseerd zijn op dezelfde recurrente betrekking voor deze kosten.

  Van deze drie functies is `losOpBU` de allerbelangrijkste: als het bottom-up dynamisch programmeren hier niet goed genoeg in te herkennen is, zal je oplossing voor de opdracht **nooit** voldoende worden, hoe goed de rest ook is. Daarnaast dient deze functie niet alleen het aantal bergen en de minimale kosten te retourneren, maar ook een verzameling bergen die hiermee overeenkomt.

- `losOpGretig` bouwt een geldige (niet per se optimale) oplossing op een gretige manier op. Dat betekent dat stap voor stap bergen aan de oplossing worden toegevoegd, op zo'n manier dat de verzameling bergen geldig blijft. Een berg die eenmaal in de verzameling zit, kan daar niet meer worden uitgehaald.

  Bedenk zelf een algoritme om de volgende berg te bepalen die je toevoegt. Dit mag een eenvoudig algoritme zijn, maar er is 0.25 punt bonus te verdienen voor de drie inzendingen bij de eerste kans van de opdracht (niet de herkansing dus) waarvoor `losOpGretig` zo groot mogelijke verzamelingen bergen oplevert voor een aantal testinstanties (en natuurlijk wel gretig is).

- `doeExperiment` vergelijkt de functie `losOpGretig` met de functie `losOpBU`. Voor een zelf te kiezen waarde van $n \geq 100$, en zinvolle waardes voor `minP`, `maxP`, `minH` en `maxH` genereert het 100 random instanties van het probleem, die allemaal met een zinvolle waarde voor het budget b worden opgelost met zowel `losOpBU` als `losOpGretig`. Het gemiddelde aantal bergen in de oplossing bij `losOpBU` en bij `losOpGretig` (gemiddeld dus over de 100 instanties) wordt berekend en op het scherm gezet.

Enkele tips bij het programmeren

- Kijk in het skeletprogramma waar allemaal `TODO` staat, voor met name de functies die geïmplementeerd moeten worden.
- Het is verstandig om eerst na te denken over hoe je de data uit een invoerbestand, de informatie over de bergen, opslaat in je klasse. Implementeer vervolgens achtereenvolgens de betrekkelijk eenvoudige memberfuncties `leesInBergen` en `drukAfInstantie`.
- Wellicht ten overvloede: als je in C++ `ifstream fin` gebruikt voor je invoer-tekstbestand, dan kun je heel eenvoudig een getal inlezen, b.v. met

  ```
  fin >> getal;
  ```

- Denk, voordat je begint met `losOpRec`, `losOpTD` en `losOpBU`, eerst goed na over de recurrente betrekking die alle drie de functies moeten gebruiken. En over hoe je bij `losOpBU` vervolgens ook de corresponderende verzameling bergen kunt afleiden, uitgaande van de tabel(len) die je vult tijdens het berekenen van de minimale kosten.
- Het lijkt handig om bij `losOpRec` en `losOpTD` een recursieve hulpfunctie te gebruiken, die als parameters o.a. waardes voor i en k meekrijgt.
- Bij diverse memberfuncties van klasse `Gebergte` moet je eerst controleren of er al een geldige instantie is (ingelezen of gegenereerd). Het is handig om daarvoor een boolean membervariabele te gebruiken die bij het inlezen of genereren een waarde krijgt.

Let op: het is niet toegestaan om de headers van de gegeven public memberfuncties in `gebergte.h` te veranderen. Dan zou onze automatische test (met een ander main programma) bij de beoordeling namelijk niet goed kunnen werken. Je mag wel functies toevoegen. Verder kan het inleveren van lelijke, niet-elegante, of erg inefficiënte code 0.5 punt aftrek opleveren.

Algemene opmerkingen

- Maak / behoud een verstandige klassenstructuur.
- Je klassen mogen alleen `public` membervariabelen en -functies hebben die buiten de klasse bekend moeten zijn. Andere membervariabelen en -functies moeten `private` zijn.
- Als er in je klassen dynamisch geheugen wordt gealloceerd, denk dan ook aan een destructor.
- Functies mogen niet te lang zijn (maximaal 35 regels).
- Gebruik constantes waar dat zinvol is. Kijk bijvoorbeeld in bestand `constantes.h` in het skeletprogramma.
- Het werkende programma mag er op het scherm eenvoudig uitzien, maar moet wel duidelijk zijn. De enige te gebruiken headerfiles zijn in principe `iostream`, `sstream`, `iomanip`, `fstream`, `cstring`, `string`, `vector`, `utility`, `set`, `unordered_set`, `cstdlib`, `ctime` en `climits`. Wil je een andere headerfile gebruiken, vraag de docent. De headerfile `algorithm` mag in ieder geval niet gebruikt worden.

- Boven elke functie in `gebergte.h` moet een commentaarblokje komen met daarin een korte beschrijving van wat de functie doet. Noem daarin tevens de gebruikte parameters: geef hun betekenis en geef aan hoe ze eventueel veranderd worden door de functie. Geef ook aan wat de memberfuncties met de membervariabelen van het object doen. Let verder op de layout (consequent inspringen) en op het overige commentaar bij de programmacode (zinvol en kort).
- Als je bij de vereiste controles in de memberfuncties een fout ontdekt, geef dan ook een passende foutmelding aan de gebruiker.
- Het programma moet onder Linux bij LIACS getest zijn en werken. Dat kan vanuit huis bijvoorbeeld op de huisuil, zie de instructie op de website.

Verslag

Het verslag moet getypt zijn in LaTeX, en moet bevatten:

- Een korte introductie, met uitleg over het probleem en de opdracht.
- Een paragraaf waarin je uitlegt wat `minKosten`$[i][k]$ voorstelt. Vervolgens behandel je de recurrente betrekkeing waar `minKosten`$[i][k]$ aan voldoet:
  - Leg uit hoe je `minKosten`$[i][k]$ kunt berekenen uit waardes van `minKosten` voor deelproblemen, en waarom dat zo is. Wees precies in je uitleg en geef zo mogelijk een formule.
  - Vermeld ook wat `minKosten`$[i][k]$ is voor basisgevallen (i, k) en waarom.
- Een paragraaf waarin je uitlegt hoe je de bergen in een optimale oplossing bepaalt. Dat wil zeggen: hoe je uit de tabel(len) die je vult tijdens `losOpBu`, terugrekent welke bergen beklommen moeten worden.
- Een paragraaf waarin je uitlegt wat de worst-case tijdcomplexiteit van je algoritme met bottom-up dynamisch programmeren is (grote O), uitgedrukt in het aantal bergen n.
- Een paragraaf waarin je uitlegt hoe je functie `losOpGretig` een oplossing stap voor stap opbouwt.
- Een paragraaf met resultaten.
  - Hier geef je voor een zelf gemaakt invoerbestand (dat je ook moet inleveren) en budget b de rekentijden van alle vier oplosmethoden, het maximaal aantal bergen dat er beklommen kan worden, de minimale kosten voor dit aantal bergen, en het aantal bergen dat er met `losOpGretig` wordt beklommen. Je moet een invoerbestand en budget kiezen waarbij alle vier de methoden binnen een aanvaardbare tijd een oplossing vinden, waarbij de rekentijd met `losOpRec` ten minste 10 seconden bedraagt. Vermeld uiteraard ook de gebruikte waarde van b.
  - Ook geef je de uitkomst van functie `doeExperiment`. Vermeld daarbij de gebruikte waarden van de parameters n, `minP`, `maxP`, `minH`, `maxH` en b, en de resulterende (gemiddelde) aantallen bergen. Welke conclusie trek je uit dit experiment.

Aanvullingen / tips / vragen

Eventuele verdere aanvullingen of tips bij de programmeeropdracht komen op de website van het vak

`https://liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/`

Daar is ook een subpagina met onder andere een template voor het LaTeX-verslag. De behaalde cijfers komen te zijner tijd in Brightspace te staan.

Heb je vragen over de opdracht, dan kun je die uiteraard stellen tijdens de practicumbijeenkomsten van het vak. Je kunt ook emailen naar algoritmiek@liacs.leidenuniv.nl

In te leveren

Via Brightspace:

- je programma (alle .cc/.h bestanden en Makefile voor Linux bij LIACS) met het invoerbestand uit het verslag,
- en een PDF van je verslag

samen in één .zip, .tgz of .tar.gz bestand. Vermeld overal duidelijk de namen van de makers. Om via Brightspace in te leveren, moet je eerst een nieuw groepje vormen voor deze opdracht (ook als je geen programmeerpartner hebt). Vervolgens kun je namens dat groepje je oplossing inleveren.

Uiterste (!) inleverdatum:
Voor maximale punten: maandag 18 mei 2026, 23.59 uur.
Met aftrek van 1 punt: dinsdag 26 mei 2026, 23.59 uur.
Met aftrek van 2 punten: maandag 1 juni 2026, 23.59 uur.

Normering:

Onderdeel:	standaard	bonus voor `losOpGretig` /
werking	5 punten	0.25 punt
commentaar en layout	1 punt	
modulaire opbouw en OOP	1 punt	
verslag	3 punten	

Het is niet toegestaan om je opdracht (code en/of verslag) te maken met behulp van een large language model als ChatGPT of DeepSeek. Bij opvallende inzendingen kunnen de makers worden uitgenodigd om hun oplossing toe te lichten.