TDDE73 – Inlämning #4

Ansvarig lärare: Christoffer Holm (`christoffer.holm@liu.se`)

Senast ändrad: 2025-12-05

Regler

- Lämna in dina lösningar till `christoffer.holm@liu.se` senast 2025–12–22 kl 23:59.
- Skicka mejlet med titeln `TDDE73: Inlämning 4`.
- Lösningar på programmeringsuppgifter ska bifogas som `.cc` och/eller `.h` filer.
- Svar på teoretiska uppgifter ska bifogas som PDF eller `.txt` filer. Datorskrivna lösningar är att föredra, men inskannade anteckningar är också OK.
- Skriv i mejlet vilka uppgifter du har lämnat in och i vilka filer de kan hittas.

Godkänd inlämning

En inlämning är godkänd om:

- Minst en uppgift per del bedöms som godkänd.
- Fyra uppgifter, eller fler, bedöms som godkända.

En inlämning kan endast kompletteras om minst fyra uppgifter lämnades in och ett försök till en lösning gjordes på vardera uppgift. Du kommer att bli inkallad till att redovisa en uppgift minst en gång under kursen.

Slutbetyg

Slutbetyg på momentet sätts enligt följande krav:

	Godkända uppgifter	**Godkända *-uppgifter**	**Samlade bonuspoäng**
Betyg 3	4 per inlämning		
Betyg 4	4 per inlämning	5 stycken totalt	
Betyg 4	4 per inlämning		20p
Betyg 5	4 per inlämning	5 stycken totalt	20p

Dessa uppgifter är nya, så de kan variera i svårighetsgrad och innehålla fel. Kontakta `christoffer.holm@liu.se` om du har frågor, funderingar eller synpunkter.

Uppgifter

Del 1 – Heap

4.1. Bevisa följande två påståenden:
(a) ett nästan komplett binärt sökträd är *inte* en heap
(b) en array sorterad i stigande ordning är en minimum heap

4.2. Implementera följande funktioner i C++, samt ange deras tidskomplexitet:
(a) funktionen `bool is_heap(std::vector<int> const& v)` som kontrollerar huruvida `v` är en heap.
(b) funktionen `trace(std::vector<int> const& v, unsigned i)` där `v` antas vara en minimum heap. `trace()` skriver ut vilka värden som besöks om man traverserar från rotnoden till noden som lagras på index `i`.
Exempel: Givet minimum heapen `v = { 1, 3, 2, 4, 6, 7, 5 }` så ska `trace(v, 5)` skriva ut `1 2 7`.

4.3.* Låt $\mathcal{H}$ vara en minimum heap av storlek n som lagras i en array. Låt $\text{rank}(i)$ vara höjden av delträdet till $\mathcal{H}$ vars rot är nod i. Notera att lövnoder har rang 0.
Givet $k \in \mathbb{N}$, låt $R_k = \{i \mid \text{rank}(i) \geq k\}$. Visa att

$$|R_k| \leq \frac{n}{2^k}.$$

Ledning: Givet att noderna definieras i termer av sina index i arrayen ger avbildningen $L(j) = 2j + 1$ det vänstra barnet till nod j. Givet i s.a. $\text{rank}(i) \geq k$, identifiera vilka noder som *måste* finnas i delträdet som har i som rot.

Del 2 – Grafsökning

OBS: I denna del antas alla grafer vara sammanhängande och ändliga.

4.4. Bestäm tidskomplexiteten för djupet-först-sökning av en graf $G = (V, E)$. Tidskomplexiteten ska uttryckas i termer av $|V|$ och $|E|$.

4.5. Konstruera en algoritm som detekterar huruvida en graf $G = (V, E)$ innehåller en cykel genom att använda grafsökning.
Förklara hur du vet att din algoritm kommer att upptäcka cykler i G. Redogör även vilken algoritm du använder som bas: djupet-först- eller bredden-först-sökning, samt varför du använder just den grafsökningsalgoritmen.

4.6. Konstruera en algoritm som detekterar huruvida en given graf $G = (V, E)$ är *bipartit*.
Förklara hur du vet att din algoritm kan avgöra om G är bipartit. Redogör även vilken algoritm du använder som bas: djupet-först- eller bredden-först-sökning, samt varför du använder just den grafsökningsalgoritmen.
Förklaring: I en bipartit graf kan V partitioneras till två mängder A och B där $A \cap B = \emptyset$ och $A \cup B = V$ sådant att för alla $(u, v) \in E$ gäller det att $u \in A$ och $v \in B$.
D.v.s. en bipartit graf är en graf där noderna kan delas upp i två distinkta mängder där varje båge har en ändpunkt i vardera mängd.

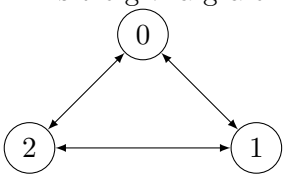
4.7.* Givet att det finns en väg mellan $u \in V$ och $v \in V$ i den oviktade grafen $G = (V, E)$, visa att bredden-först-sökning garanterat hittar den *kortaste* vägen från u till v.

Del 3 – Grafalgoritmer

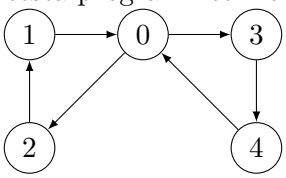
4.8. En Eulerkrets är en krets som besöker varje båge i en graf. En riktad graf $G = (V, E)$ har en Eulerkrets om och endast om varje nod $v \in V$ har lika många inkommande som utgående bågar. D.v.s. att

$$|\{(u, v) \in E \mid u \in V\}| = |\{(v, u) \in E \mid u \in V\}|.$$

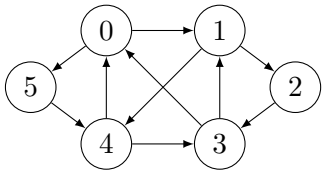
I `givna_filer/euler.cc` implementeras funktionen `has_euler_circuit()` som kontroller villkoret ovan för en graf. Dessutom implementeras `find_euler_circuit()` som hittar en Eulerkrets givet att en sådan finns. Båda dessa funktioner nyttjar den inkompletta graf-representationen `Graph`.
Din uppgift är att implementera `Graph` klassen genom att välja en lämplig underliggande representation (grannlista eller grannmatris) och implementera alla funktioner. Huvudprogrammet, `has_euler_circuit()` och `find_euler_circuit()` ska ej modifieras.
Notera att grafen är *riktad* så nod (u, v) och (v, u) är olika bågar för $u, v \in V$. Detta måste fångas i `Graph`. Notera specifikt att `out_degree(v)` och `in_degree(v)` representerar antalet utgående och inkommande bågar, respektive, för noden `v`.
Det finns tre givna grafer du kan testa programmet mot. Dessa kan ses nedan:



`graph1.txt`

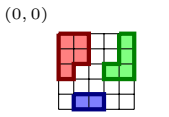


`graph2.txt`

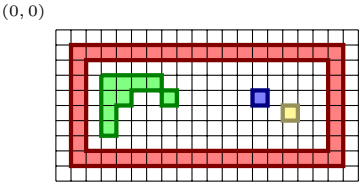


`graph3.txt`

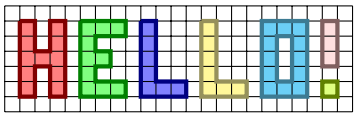
4.9. I filen `givna_filer/cluster.cc` finns början på ett program som läser in ett ändligt rutnät av `bool` värden. Rutnätet innehåller *kluster* av celler som är `true`. Ett kluster definieras som en sammanhängande uppsättning av celler som alla är `true`. Två celler är grannar om de är ortogonalt eller diagonalt jämte varandra.
Programmet läser in ett rutnät från en fil. Det finns tre filer givna, dess innehåll kan ses i diagrammet nedan:



`grid1.txt`



`grid2.txt`


`grid3.txt`

Notera att i diagrammet är varje sammanhängande kluster färgade i samma färg för att tydligare visa vart klustret är. Ofärgade rutor motsvarar celler som är `false`. Rad 0 motsvarar översta raden, och kolumn 0 motsvarar första kolumnen från vänster.
Din uppgift är att implementera funktionen `find_region_size()` så att den, givet ett rutnät `grid` och en position `start`, hittar antalet `true`-celler som `start` är sammanhängande med (inklusive sig själv). Om `start = { 0, 0 }` i `grid1.txt` ska `find_region_size()` returnera `5` ty det är exakt antalet celler i den röda regionen (som $(0, 0)$ råkar ligga i). Funktionen returnerar `0` för alla celler som är `false`.
Till din hjälp finns funktionerna `get_neighbours()` som returnerar alla *potentiella* grannar till en cell, och `in_bounds()` som kontrollerar huruvida en cell faktiskt ligger i rutnätet.

4.10.* I denna uppgift ska du skriva ett program som hittar den kortaste *ordkedjan* från en sträng till en annan. Början av en implementation finns i `givna_filer/chain.cc`.
En ordkedja i denna uppgifter definieras som en sekvens av *giltiga* ord $w_1, w_2, \ldots, w_n$ där w_i och w_{i+1} skiljer sig som mest på *en* position. T.ex. skiljer sig orden "word" och "ward" endast på den andra position i orden, övriga position innehållar samma bokstäver.
Ett ord räknas som *giltigt* om det förekommer i filen `givna_filer/words.txt`.
Ett exempel på en giltig ordkedja är:

$$\text{code} \rightarrow \text{mode} \rightarrow \text{made} \rightarrow \text{mate} \rightarrow \text{math}$$

där vi ser att vardera ord endast skiljer sig på en position jämfört med föregående ord. Detta är också ett exempel på en *kortaste* ordkedja mellan code och math, d.v.s. att det finns ingen annan ordkedja mellan dessa ord som är kortare.
Din uppgift är att färdigställa funktionen `wordchain()` genom att göra en lämplig grafsökning från ordet `start` till ordet `target`. För att avgöra om ett ord är giltigt tar funktionen emot ett lexikon med alla giltiga ord.
För att göra grafsökningen har vi en *implicit* båge från ett giltigt ord w till alla andra giltiga ord som skiljer sig på exakt en position jämfört med w. En implicit båge syftar till att vi inte behöver representera grafstrukturen i minnet utan vi kan istället beräkna alla *potentiella* grannar till ett ord när de behövs med hjälp av funktionen `get_candidates()`. Vi måste dock själva dubbelkolla vilka ord som funktionen producerade som var giltiga.
För att hålla koll på den kortaste vägen från `start` till ett annat ord `word` använder vi strukturen `parents` som lagrar `word` som nyckel och ordet vi kom ifrån som värde. T.ex. skulle vi lagra följande värden (och potentiellt fler) för ordkedjan ovan som:

```
parents["code"] = ""
parents["made"] = "mode"
parents["mate"] = "made"
parents["math"] = "mate"
parents["mode"] = "code"
```

Formelsamling

Antag att $a > 0$, $b > 0$ och $f, g, h : \mathbb{N} \to \mathbb{R}_+$ s.a. f, g, h är obegränsade uppåt.

Definitioner:

(i) $g(n) \in \mathcal{O}(f(n))$ om och endast om $\exists c, n_0 > 0$ sådana att $g(n) \leq cf(n)\ \forall n \geq n_0$.

(ii) $g(n) \in \Omega(f(n))$ om och endast om $\exists c, n_0 > 0$ sådana att $g(n) \geq cf(n)\ \forall n \geq n_0$.

(iii) $g(n) \in \Theta(f(n))$ om och endast om $g(n) \in \mathcal{O}(f(n))$ och $g(n) \in \Omega(f(n))$.

(iv) $\mathcal{O}(g(n)) \subset \mathcal{O}(f(n))$ om och endast om:

- $h_1(n) \in \mathcal{O}(f(n))$ för alla $h_1(n) \in \mathcal{O}(g(n))$
- Det finns $h_2(n) \in \mathcal{O}(f(n))$ sådan att $h_2(n) \notin \mathcal{O}(g(n))$

Om andra kravet inte är uppfyllt gäller $\mathcal{O}(g(n)) \subseteq \mathcal{O}(f(n))$.

Relationer:

(v) $g(n) \in \Omega(f(n)) \iff f(n) \in \mathcal{O}(g(n))$

(vi) $g(n) \in \Theta(f(n)) \iff g(n) \in \mathcal{O}(f(n))$ och $f(n) \in \mathcal{O}(g(n))$

Räkneregler:

(vii) $a \cdot g(n) + b \in \mathcal{O}(f(n)) \iff g(n) \in \mathcal{O}(f(n))$

(viii) $a \cdot g(n) + h(n) \in \mathcal{O}(f(n)) \iff g(n), h(n) \in \mathcal{O}(f(n))$

(ix) $g(n)h(n) \in \mathcal{O}(f(n)) \implies g(n) \in \mathcal{O}(f(n))$ och $h(n) \in \mathcal{O}(f(n))$

(x) $\mathcal{O}(a \cdot f(n) + b) = \mathcal{O}(f(n))$

(xi) $\mathcal{O}(a \cdot f(n) + g(n)) = \mathcal{O}(f(n)) \iff \mathcal{O}(g(n)) \subseteq \mathcal{O}(f(n))$

(xii) $\mathcal{O}(h(n)f(n)) \subseteq \mathcal{O}(g(n)f(n)) \iff \mathcal{O}(h(n)) \subseteq \mathcal{O}(g(n))$

Komplexitetsklasser:

(xiii) $\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log n) \subset \mathcal{O}(2^n)$

(xiv) $\mathcal{O}(\log_\alpha n) = \mathcal{O}\left(\log_\beta n\right) = \mathcal{O}(\log n)$ för alla $\alpha, \beta \in \mathbb{R}_+$.

(xv) $\mathcal{O}(n^\alpha) \subset \mathcal{O}\left(n^\beta\right)$ för alla $0 < \alpha < \beta$

(xvi) $\mathcal{O}(\alpha^n) \subset \mathcal{O}(\beta^n)$ för alla $0 < \alpha < \beta$.

(xvii) $\mathcal{O}(n \log n) \subset \mathcal{O}\left(n^{1+\varepsilon}\right)$ för alla $\varepsilon > 0$

(xviii) För något $k > 0$ och $c \in \mathbb{R}_+$ så gäller $g(n) \leq c\ \forall n \geq k \iff g(n) \in \mathcal{O}(1)$

Användbara samband:

(xix) $\log_a x \leq x$ då $x \geq 3$ $(x \geq e)$ och $a \geq 2$ $(a \geq e^{1/e})$

(xx) $n \leq n^k$ då $n \geq 1$ och $k \geq 1$

Tidskomplexitet

Låt $\vec{n}$ representera en vektor med alla variabler som påverkar tidskomplexiteten.

[I] Följande operationer antas ha tidskomplexitet $\mathcal{O}(1)$:

- Aritmetiska och logiska operationer
- Initiering och tilldelning av inbyggda datatyper
- Avreferering av pekare och åtkomst av variabler
- Allokering och avallokering av minne

[II] Givet två satser som evalueras i följd, med tidskomplexitet $\mathcal{O}(f_1(n))$ och $\mathcal{O}(f_2(n))$ respektive, så är totala tidskomplexiteten $\mathcal{O}(f_1(n) + f_2(n))$.

[III] Villkorssatser, t.ex. `if`- eller `switch`-satser, har tidskomplexitet $\mathcal{O}(g(\vec{n}) + f(\vec{n}))$ där $f(\vec{n})$ är tidskomplexiteten av den dyraste (billigaste om bästa fall) grenen i villkorssatsen och $g(\vec{n})$ är tidskomplexiteten av villkoret.

[IV] Upprepningssatser, t.ex. `for`-, `while`- och `do-while`-loopar, har tidskomplexitet givet av $\mathcal{O}(g(\vec{n}) \cdot f(\vec{n}))$ där $g(\vec{n})$ är maximala (minimala om bästa fall) antalet iterationer som kan ske och $f(\vec{n})$ är tidskomplexiteten av den dyraste (billigaste om bästa fall) iterationen.

Rekursiva komplexitetsklasser

Givet en rekursiv algoritm där:

- det sker $a \geq 1$ rekursiva anrop
- indatan delas upp i $b > 1$ lika stora bitar
- det tar $f(n) \in \mathcal{O}\left(n^d\right), d \geq 0$ tid att dela upp och kombinera problemet

Så ges tidskomplexiteten av differensekvationen

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

och den slutgiltiga tidskomplexiteten bestäms av följande:

[V] Om $d < \log_b a \implies T(n) \in \Theta\left(n^{\log_b a}\right)$

[VI] Om $d = \log_b a \implies T(n) \in \Theta\left(n^d \log n\right)$

[VII] Om $d > \log_b a \implies T(n) \in \Theta\left(n^d\right)$

Specialfall av [VI]:

[VIII] Om $a = 1$ och $d = 0 \implies T(n) \in \Theta(\log n)$

Amorterad tidskomplexitet

[IX] Antag att algoritmen körs M gånger och den *totala* tidskomplexiteten för alla M körningar är $\mathcal{O}\left(T(\vec{n}, M)\right)$. Då ges den *amorterade tidskomplexiteten* av

$$\frac{\mathcal{O}\left(T(\vec{n}, M)\right)}{M} = \mathcal{O}\left(\frac{T(\vec{n}, M)}{M}\right).$$

Heaps

Givet en heap $\mathcal{H}$ med jämförelseoperator $\leq_{\mathcal{H}}$ som är byggd ovanpå en dynamisk array, för index $i = 0, 1, \ldots, n$ – där $\mathcal{H}_i$ motsvarar det lagrade värdet – så gäller:

(xix) $P(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$ ger föräldern till noden på plats i

(xx) $L(i) = 2i + 1$ ger vänsterbarnet till noden på plats i

(xxi) $R(i) = 2i + 2$ ger högerbarnet till noden på plats i

(xxii) $\mathcal{H}_i \leq_{\mathcal{H}} \mathcal{H}_{L(i)}$ om $L(i) \leq n$ och $\mathcal{H}_i \leq_{\mathcal{H}} \mathcal{H}_{R(i)}$ om $R(i) \leq n$.

Grafer

[XXIII] En graf $G = (V, E)$ är en mängd *noder* V och en mängd *bågar* E där $E \subseteq V \times V$.

[XXIV] $G = (V, E)$ är *ändlig* om både E och V är ändliga mängder.

[XXV] $G = (V, E)$ är *oriktad* om paren i E är *oordnade*, d.v.s. att alla bågar saknar riktning.

[XXVI] $G = (V, E)$ är *riktad* om paren i E är *ordnade*, d.v.s. att alla bågar har en riktning.

[XXVII] En *viktad* graf $G_w = (V, E)$ är en graf där det existerar en avbildning $w : E \to \mathbb{R}$, d.v.s. i en viktad graf har varje båge ett associerat värde som kallas dess *vikt*.

[XXVIII] En *vandring* i en graf $G = (V, E)$ är en sekvens av noder där det finns en båge från föregående nod till nästa nod i sekvensen.

[XXIX] En *väg* är en vandring som inte återupprepar bågar.

[XXX] En *stig* är en vandring som inte återupprepar noder.

[XXXI] En *krets* är en väg som börjar och slutar i samma nod.

[XXXII] En *cykel* är en stig som börjar och slutar i samma nod.

[XXXIII] $G = (V, E)$ är *sammanhängande* om det finns en väg mellan varje par av noder.

Användbara samband:

För en graf $G = (V, E)$ gäller:

(xix) Om G är oriktad och sammanhängande: $|V| + 1 \leq |E| \leq \frac{|V|(|V|-1)}{2}$

(xx) Om G är riktad och sammanhängande: $|V| + 1 \leq |E| \leq |V|(|V| - 1)$

(xxi) Om G är riktad: $\deg^+(v) + \deg^-(v) = \deg(v)$ för alla $v \in V$