Uitwerking tentamen Algoritmiek, donderdag 9 juni 2022



1(a)

```
int kostenHamiltonkring (int n, int route[])
{ bool gehad[n];    // hebben we knoop i al gehad
  int kosten;

  for (int i=0; i<n; i++)
    gehad[i] = false;
  if (route[0] != route[n])  // kring sluit niet.
    return -1;

  kosten = 0;
  for (int pos=0; pos<n; pos++)
  { i = route[pos];
    j = route[pos+1];
    if (gehad[i])  // knoop i staat al eerder in rijtje
      return -1;
    if (adj[i][j] == 0)  // tak bestaat niet
      return -1;
    gehad[i] = true;
    kosten += adj[i][j];
  } // for
  return kosten;
} // kostenHamiltonkring
```

10.40

(b)

```
void bepaalRoute (int n, int route[], int pos, int besteRoute[], int &best)
  { int kosten;
    if (pos == n)  // sluit de kring
    { route[pos] = route[0];    // 0 dus.
      kosten = kostenHamiltonkring (n, route);
      if (kosten != -1 && kosten < best)  // betere route!
      { for (int p=0; p≤n; p++)
           besteRoute [p] = route[p]
        best = kosten;
      }
    }
    else  // pos < n
    { if (pos == 0)
      { route [pos] = 0;
        bepaalRoute (n, route, pos+1, besteRoute, best);
      }
      else  // 1 ≤ pos < n ⇒ probeer alle knopen (zelfs 0!)
      { for (int i=0; i<n; i++)
        { route [pos] = i
          bepaalRoute (n, route, pos+1, besteRoute, best);
        }
      }
    }
  } // bepaalRoute
```

10.52 / 10.54 / 10.56

(c) De functie genereert alle rijtjes route[0], route[1], ..., route[n-1], route waarbij route[0] = route[n] = 0. Voor elk van route[1], ..., route[n-1] zijn dan n mogelijkheden. Gecombineerd geeft dat $n^{n-1}$ rijtjes.
Voor elk van die rijtjes wordt kostenHamiltonkring aangeroepen.
Die functie heeft complexiteit in $\Theta(n)$.
De totale complexiteit wordt dus $\Omega(n * n^{n-1}) = \Omega(n^n)$.

1.02 $\Omega$, want je zou kunnen zeggen dat alle deelrijtjes ook meetellen voor de complexiteit. Dan krijg je

$$\underbrace{1}_{pos=0} + \underbrace{n}_{1} + \underbrace{n^2}_{2} + \cdots + \underbrace{n^{n-1}}_{n-1} + \underbrace{n^{n-1}}_{n} = \frac{n^n - 1}{n-1} + n^{n-1}$$ deelrijtjes in totaal.

Ah, dat is nog steeds in $\Theta(n^{n-1})$, en met aanroep van bepaalKosten-Hamiltonkring komen we dan in $\Theta(n^n)$.

11.08

(d)

```
route [0] = 0;
for (pos = 1; pos < n; pos++)
{ vind buurknoop j van route [pos-1],
    die nog niet in route [0],..., route [pos-1] voorkomt,
    zodat adj [route [pos-1]] [j] minimaal is.
  if (zo'n buurknoop j bestaat)
    route [pos] = j;
  else // alle buurknopen van route [pos-1] zitten al in de route
    return -1; // geen oplossing gevonden
}
if (adj [route [n-1]] [0] != 0) // sluit de kring
{ route [n] = 0;
  return bepaalKostenHamiltonkring (n, route);
}
else // niet mogelijk om de kring te sluiten
  return -1;
```

11.16

Voor de voorbeeldgraaf:

route [0] = 0
route [1] = 2
route [2] = 1
route [3] = 4
route [4] = 5
route [5] = 7
route [6] = 6
route [7] = 3
return -1; // we kunnen de kring niet sluiten
$\Rightarrow$ geen geldige Hamiltonkring.

11.21

2(a)

Uitwerking tentamen Algoritmiek, donderdag 9 juni 2022

2 (a)

```
int vindPositie1 (int A[], int n, int x)
{
  for (int i=0; i<n; i++)
  { if (A[i] == x)
      return i;
  }
  return -1;   // niet gevonden
}
```

(b) 11.25 / 11.26 / 11.27

```
int vindPositie2 (int A[], int links, int rechts, int x)
{ // pre: links ≤ rechts
  int mid,    // 'midden' van deelarray, of eigenlijk: precies links van midden
      pos;

  if (links == rechts)
  { if (A[links] == x)
      return links;
    else
      return -1;
  }
  else   // links < rechts
  { mid = (links + rechts) / 2;
    pos = vindPositie2 (A, links, mid, x);
    if (pos == -1)  // links niet gevonden, zoek rechts
      pos = vindPositie2 (A, mid+1, rechts, x);
    return pos;
  }
}
```

11.35

(c)

```
int vindPositie3 (int A[], int n, int x)
{ int links, rechts,   // huidige grenzen
      mid;

  links = 0;
  rechts = n-1;
  while (true)
  { if (links > rechts)   // kan gebeuren !
      return -1                 b.v. als we x = 3 zoeken
    else                        in array met n=2 elementen
                                A[0] = 4, A[1] = 5
```

```
    { if (links == rechts)        // dit geval zouden we over kunnen
      { if (A[links] == x)        // slaan, en direct met mid beginnen
          return links;
        else
          return -1;
      }
      else  // links < rechts
      { mid = (links + rechts)/2;
        if (A[mid] == x)
          return mid;
        else
          if (A[mid] > x)         // x kan links van mid staan
            rechts = mid-1
          else   // A[mid] < x
            links = mid + 1;
      }
    }  // else
  }  // while
} // vindPositie 3
```

11.46 / 11.48

(d) Binair zoeken is een voorbeeld van 'decrease-by-a-constant-factor-and-conquer. Iedere iteratie van de while-lus wordt de grootte van het deelarray dat we nog moeten bekijken grofweg gehalveerd.

11.50 / 11.56

3 (a)

Basisgeval

Als $i=0$ en/of $j=0$, dan kijken we naar de eerste 0 letters van A en/of B. Daar kunnen we geen substring van lengte $\geq 1$ in vinden. Alleen lengte 0 is dus mogelijk.

Recursieve stap:

Als $i, j \geq 1$, dan is $A[i-1]$ de laatste letter van A die we bekijken en is $B[j-1]$ de laatste letter van B die we bekijken.

Als $A[i-1] = B[j-1]$, dan kunnen we het beste deze twee letters aan elkaar koppelen. Dat levert vast 1 letter op voor de gemeenschappelijke substring. Vervolgens moeten we nog zoveel mogelijk van de eerste $i-1$ letters van A zien te koppelen aan de eerste $j-1$ letters van B. Dat levert nog eens $L(i-1, j-1)$ op.

Totaal $1 + L(i-1, j-1)$ dus. Merk op dat we $A[i-1]$ en $B[j-1]$ maar aan één letter mogen koppelen.

12.07/12.09

Als A[i-1] ≠ B[j-1], dan kunnen we in ieder geval niet die twee letters aan elkaar koppelen.
Misschien kunnen we B[j-1] wel aan een van de eerdere letters van A koppelen. Dan hebben we te maken met L(i-1, j). (eerste i-1)
Misschien kunnen we A[i-1] wel aan een van de eerdere (eerste j-1) letters van B koppelen. Dan hebben we te maken met L(i, j-1).
Van deze twee kiezen we (natuurlijk) het maximum, want we zoeken de langste gemeenschappelijke substring.
(Dit werkt ook goed als we A[i-1] en B[j-1] aan geen enkele letter in de andere string kunnen koppelen)

12.14
(b)

| L(i,j) | | | f | i | e | t | s |
|---|---|---|---|---|---|---|---|
| | | j=0 | 1 | 2 | 3 | 4 | 5 |
| | i=0 | 0 | 0 | 0 | 0 | 0 | 0 |
| l | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| i | 2 | 0 | 0 | 1 | 1 | 1 | 1 |
| f | 3 | 0 | 1 | 1 | 1 | 1 | 1 |
| t | 4 | 0 | 1 | 1 | 1 | 2 | 2 |

12.17/12.19
(c)

```
int overlap (string A, int m, string B, int n)
{ int L[m+1][n+1];
  for (int j=0; j<=n; j++)   // vul bovenste rij van L
    L[0][j] = 0;

  for (int i=1; i<=m; i++)   // vul rij i van L
  { L[i][0] = 0;
    for (int j=1; j<=n; j++)
    { if (A[i-1] == B[j-1])
        L[i][j] = 1 + L[i-1][j-1];
      else
      { if (L[i-1][j] > L[i][j-1])
          L[i][j] = L[i-1][j];
        else
          L[i][j] = L[i][j-1];
      }
    } // for j
  } // for i
  return L[m][n];
}
```

226

(d)

De tijdcomplexiteit van de functie overlap is $\Theta(m \times n)$

Een basisoperatie is de test

if (A[i-1] == B[j-1])

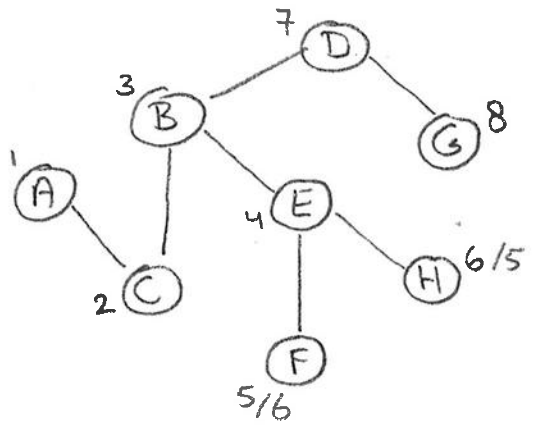
Die wordt iedere iteratie van de for-j-lus binnen iedere iteratie van de for-i-lus uitgevoerd. In totaal dus $m \times n$ keer

12.30/12.3~~6~~7

4) a) Een minimale opspannende boom van G is een boom, bestaande uit takken van G, die alle knopen van G bevat (dat is een opspannende boom van G), met minimaal totaal gewicht (som van de gewichten van alle takken in de boom).

12.40/12.45/12.49/12.~~52~~ 50

b i) Mijn klad uitwerking van het algoritme van Prim:




Nummers bij de knopen geven de volgorde van toevoeging aan de boom aan.

Mogelijke volgordes: A C B E F H D G ⟹ volgorde 2

A C B E H F D G ⟹ volgorde 4

12.57

b ii) De boom hierboven is de enige goede bij bovenstaande pseudocode. Knoop D krijgt tak-waarde 6 na toevoegen van B aan de boom. Als vervolgens E aan de boom wordt toegevoegd, levert dat geen betere takwaarde voor D op. De kandidaattak voor D blijft dus (B,D).

Dat is dus boom 3.

13.01