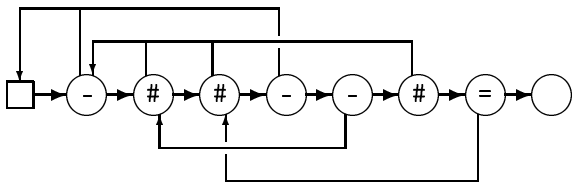
KTH, NADA, Erik Forslin

2D1320, Tilda, Tentamenslösning 6 april 2002

(5p) **1.** *Båtautomatik*



i	next[i]
1	0
2	1
3	1
4	0
5	2
6	1
7	3

2. *Båtbogsering*

(1p) Basfall:
Om rotbåten inte bogserar något är längden 5.

(4p) Allmänt fall/rekursionsfall:
Längden av hela bogsersläpet är maximala längden av vänster och höger delsläp plus 25 meter för rotbåten (med linor).

3. *Båtstuvning*

(5p) Ta hand om varje båt efter klassificeringsstationen. Upprepa följande för samtliga inkommande båtar: om startgruppsnumret hör till högsta startgruppen, skicka båten till stacken/hamnen, annars till kön/slingan runt ön. Placera nu hamnchefens båt i kön som en markör. Snurra igenom kön till dess markören dyker upp samtidigt som alla båtar med nästa lägre startgruppsnummer placeras i stacken. Upprepa förfarandet för alla lägre startgruppsnummer tills kön är tom. Nu är stacken/hamnen rätt ordnad!

4. *Båtsortering*

(2p) Insättningssortering har komplexitet $O(N^2)$ men mycket snabb då databasen är nästan sorterad. Inget extra minne behövs vid sorteringen. Antalet jämförelser då databasen är helt osorterad är ungefär $500^2 = 250\,000$. Principen för insättningssortering är att plocka ett element, skyffla bak varje element som ska ligga efter det nya för att fixa en ledig plats samt sätta in det nya elementet. Detta görs för varje element (därav en faktor N i komplexiteten) och varje insättning kan i värsta fall göra att alla sorterade element måste skyfflas bak (därav en faktor N till i komplexiteten). Är det nästan sorterat skyfflas färre element vid varje insättning och därmed minskar den andra faktorn. Är datat helt sorterat går insättningssortering i $O(N)$.

(2p) Quicksort har komplexitet $O(Nlog_2N)$ och den snabbaste i allmänna fall. Inget extra minne behövs vid sorteringen. Antalet jämförelser då databasen sorteras är ungefär $1.4 \times 500log_2500 = 6\,300$. Faktorn 1.4 kommer av att quicksort ibland plockar ett dåligt gränsvärde som ger upphov till sneda partitioner vid sorteringen. Principen för quicksort är att plocka ett gränsvärde (pivot), köra damernaförst

för att dela upp datat i två partitioner och sedan köra quicksort på respektive partition. Damerna först går i $O(N)$ och måste göras $log_2 N$ gånger eftersom vi varje gång delar datat i två ungefär lika stora partitioner. Andra faktorn närmar sig N om partitionerna blir väldigt snedfördelade. Värstafallskomplexiteten för quicksort är alltså $O(N^2)$.

(2p) Insättningssortering är bäst här! Eftersom de sorterar lite då och då och båtarna ligger på ungefär samma platser vid varje sortering kommer inssättningssorteringen bli mycket snabb. Quicksort har faktorn 1.4×9 inblandad och insättningssortering en faktor som motsvarar det maximala antal placeringar som en båt ändras mellan sorteringarna (vilket är ett litet tal).

5. *Båtflytt*

(8p) Använd bästaförstsökning med en prioritetskö som prioriterar på lägsta seglade totaltiden. Låt varje nod innehålla total seglingstid samt en faderspekare (för rekursiv utskrift av vägen då lösning hittats). Princip för trädsökningen:

1. Lägg startpunktsnoden med totaltiden noll och ingen faderspekare i priokön.
2. Upprepa punkterna 3–4 så länge kön inte är tom.
3. Plocka ut en fadersnod ur kön. Om detta är slutpunkten, skriv ut vägen rekursivt och avsluta.
4. Generera en son i taget genom att för varje punkt runt omkring fadersnoden skapa en sonnod med seglingstiden ökad beroende på vindstyrka, vindriktning och placering i förhållande till fadersnoden. Lägg in sonnoden i priokön.

   Om dumsonsträd ska utnyttjas måste det ta hänsyn till både punkten och totala seglingstiden till den punkten. Alla söner med sämre tider till samma punkt är då dumsöner.

   Eventuellt krävs någon snabb uppslagning av punkternas information, t ex med en hashtabell som hashar på punkternas nummer eller position.

(2p) För kortaste vägen, använd istället bredden först med en vanlig kö och blanda inte in totala seglingstiden i varje nod.

(6p) **6.** *Båtkommentator*

```
<sändning>  ::= <kommentar> | <kommentar> <sändning>
<kommentar> ::= <händelse>. | <händelse> utanför <plats>.
<händelse>  ::= <båtlista> passerar nu <båtlista>
<båtlista>  ::= <båt> | <båtar>
<båtar>     ::= <båt> och <båt> | <båt>, <båtar>
<båt>       ::= Assa Abloy | Djuice | Ericsson | ...
<plats>     ::= Almagrundet | Gotska Sandön | Fårö | ...
```

7. *Båtsökning*

(2p) Linjärsökning har komplexiteten $O(N)$ och fungerar lika bra med en länkad lista eller kö som med en vektor. Datat behöver inte vara sorterat. Jämförelser i medel

blir $500/2 = 250$ och i värsta fallet 500. Fördelarna är att den är enkel att förstå, fungerar lika bra med listor och kan användas då datat är osorterat. Nackdelen är att den är långsam.

(2p) Binärsökning har komplexiteten $O(log_2 N)$ och fungerar både med en sorterad vektor och ett binärt sökträd. Jämförelser då det sökta finns blir $log_2 500 \approx 9$ och då det inte finns 10. Fördelarna är att den enkelt kan skrivas rekursivt och är relativt snabb. Nackdelen i fallet träd är att sökningen går långsammare då trädet är obalanserat och i fallet vektor att den måste vara sorterad.

(2p) Hashning har komplexiteten $O(1)$ och utnyttjar en hashtabell (vektor med krocklistor). Antalet jämförelser är drygt 1 på grund av eventuella krockar. Fördelen är att den är oerhört snabb, nackdelarna är att hashtabellen måste göras något större än antalet element som ska lagras och att hashfunktionen måste sprida ut elementen väl (båda för att minska antalet krockar) samt att hashfunktionen kan ta lång tid på sig vid uträknandet av hashvärdet. Specialfallet bloomfilter kan också vara en enorm fördel då den går snabbt, tar lite minne och lagrar datat krypterat.

(1p) Hashning är snabbast. Den är ungefär 9 gånger snabbare än binärsökning och 250 gånger snabbare än linjärsökning.

(1p) Beskrivning av hashning kan hittas i föreläsningsanteckningarna.

8. *Båtvård*

(3p) Abstraktion är bra eftersom:

- Användaren behöver inte bry sig om detaljer. En abstrakt datatyp avslöjar inte lagringssättet.
- Användaren kan inte missbruka detaljinformation och därmed inte misstolka konstruktörens avsikter. Konstruktören kan specificerar operationer för åtkomst och ändring.
- Konstruktören kan anpassa den abstrakta datatypen/algoritmen till olika användare utan förstöra en bra konstruktion.
- Konstruktören kan förbättra konstruktionen utan att användarens användnig behöver ändras.

(2p) Ett abstrakt `Segel` kan vara en klass med till exempel följande innehåll.

```
Segel(); // Konstruktor
int compareTo(Segel segel2);
double getSqFeet();
double getSqMeter();
void setSqFeet(double area);
void setSqMeter(double area);
boolean equals(Segel segel2);
```