ERHVERVSAKADEMI ÅRHUS

Del, løs og kombinér

[Del, løs og kombinér]

1. Indledning

En ofte anvendt måde at løse problemer på, er ved hjælp af trinvis forfinelse, hvor vi deler et problem op i mindre delproblemer, som kan løses hver for sig. I denne note vil vi specielt betragte situationen, hvor nedbrydningen af et problem giver mindre problemer af samme type som det oprindelige. Sådanne problemer kaldes rekursivt nedbrydelige. Vi vil i det følgende formulere en algoritmeskabelon for løsning af rekursivt nedbrydelige problemer. Teknikken, som skabelonen beskriver, kaldes del, løs og kombinér.

1.2 Del, løs og kombinér skabelonen

Skal være rekursivt nedbrydeligt for at anvende del, løs og kombinér

Som nævnt i indledningen skal et problem være rekursivt nedbrydeligt, for at man kan anvende del, løs og kombinér skabelonen til formulering af en løsning herpå. Et problem er rekursivt nedbrydeligt, hvis det:

- Enten
  - Er simpelt og kan løses direkte
- Eller
  - Ikke er simpelt, men kan nedbrydes i (typisk to) delproblemer, der hver især er simplere end det oprindelige, og hvis løsninger kan kombineres til en løsning på det oprindelige problem.

Simplere betyder i denne sammenhæng, at nedbrydningen til sidst resulterer i simple problemer.

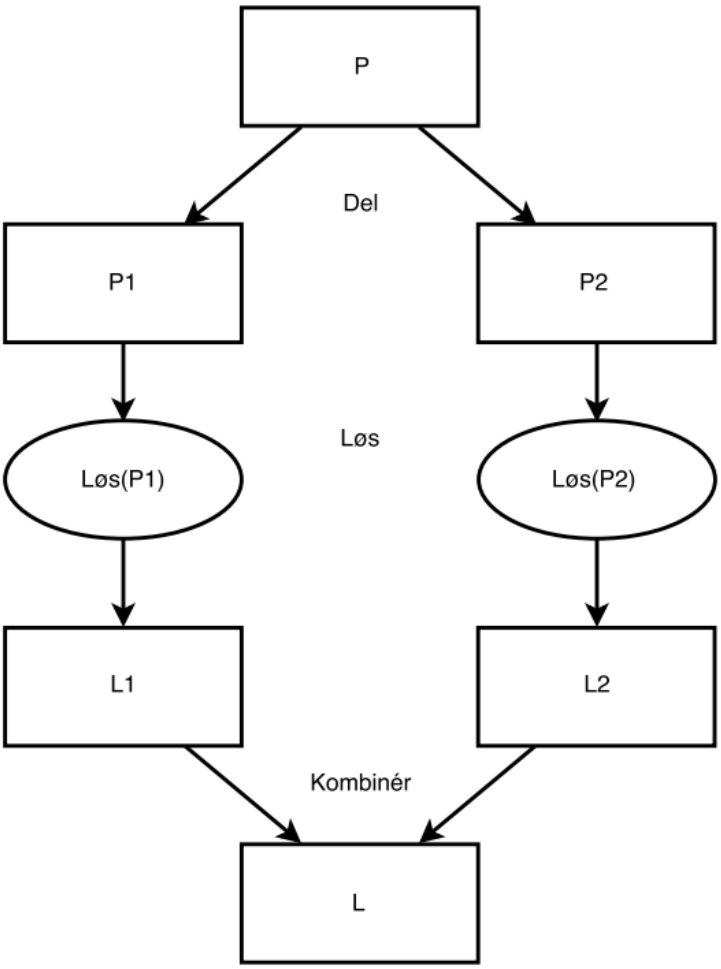
Eksempel

Summering af elementerne i en liste er et rekursivt nedbrydeligt problem. Simple problemer er en del af listen med et element. Ikke simple problemer er dele af listen med flere elementer (for eksempel hele listen). En liste med flere elementer kan nedbrydes i to dele med halvt så mange elementer, og det er klart, at nedbrydningen til sidst resulterer i simple problemer.

Algoritmer, konstrueret over del, løs og kombinér skabelonen, kan abstrakt beskrives som følger:

```
Metode: Løs(P)
Hvis p er simpelt, så returner L = løsning
Ellers
    Opdel P i P1 og P2
    L1 = Løs(P1)
    L2 = Løs(P2)
    L = Kombinér(L1,L2)
    Returner L;
```

Idéen i skabelonen er, at et ikke-simpelt problem løses ved at opdele det i to simplere delproblemer, som løses rekursivt. Derefter kombineres løsningerne på de to delproblemer til en løsning på det oprindelige problem. Figuren nedenfor illustrerer dette:



Bemærk skabelonens rekursive natur, som naturligvis hænger sammen med, at den anvendes på rekursivt nedbrydelige problemer. Strengt taget er det ikke nødvendigt at formulere skabelonen (eller konkretiseringer heraf) rekursivt; men hvor den rekursive formulering er elegant og simpel, vil en iterativ formulering typisk være kluntet og uoverskuelig.

Brug af skabelonen resulterer ofte i let formulerede og effektive algoritmer til løsninger af klassiske problemstillinger – blandt andet flere effektive sorteringsalgoritmer.

I praksis lader man sjældent et problem blive nedbrudt i mere end to delproblemer, som det fremgår af skabelonen. Principielt er der dog ikke noget til hinder for at dele op i flere.

1.3 Konkretiseringer af del, løs og kombinér skabelonen

I det følgende vil vi illustrere, hvorledes problemer kan løses som konkretiseringer af skabelonen.

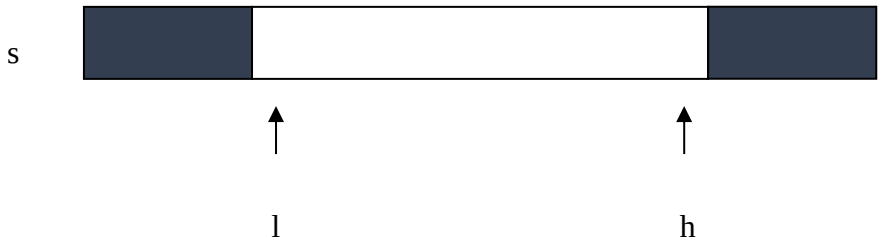
Det første problem er af en sådan natur, at det trivielt kan løses iterativt – endda med en bedre pladskompleksitet til følge. Det hænger sammen med, at eksemplet er så simpelt, som det er. Pointen med dette eksempel er imidlertid blot at illustrere teknikken på et i forvejen kendt eksempel.

Efterfølgende kommer eksempler med sortering, hvor teknikken giver en såvel effektiv som simpel algoritme.

1.4 Beregning af største tal i en Liste.

I nedenstående eksempel illustreres, hvorledes skabelonen kan anvendes til beregning af det største tal i en List. Idéen er at opdele listen i to dele, og beregne maksimum for hver af disse (to rekursive problemer af samme type som det oprindelige, men hver især kun halvt så store). Når maksimum af de to dele er fundet kan maksimum af hele listen findes, ved at returnere det største af de to delresultater.

Opdelingen af en liste i dele kan fx gøres ved at lade to indeks udpege den del af listen, der er relevant i den aktuelle sammenhæng:



En rekursiv metode der løser problemet at finde det største tal i en liste, kan da gives ved:

```
/**
 * Returnerer maksimum i s[l..h]
 * Krav: l <= h
 */
private int maximum(ArrayList<Integer> list, int l, int h) {
      if (l == h) {    kigger kun på 1 element <— kun én lang
            return list.get(l);
      } else {
            int m = (l + h) / 2;
            int max1 = maximum(list, l, m);    går fra low til midt
            int max2 = maximum(list, m + 1, h);    går fra middle + 1 til h
            if (max1 > max2) {
                  return max1;
            } else {
                  return max2;
            }
      }
}
```

For at anvender af metoden, ikke ved start skal angive, hvilken del af listen der skal findes maksimum for, kan man lade den rekursive metode være private, og i stedet sætte en public metode til rådighed. Denne metode får hele listen med som parameter og kalder den rekursive private metode, der løser problemet. Den private rekursive metode kaldes også for en rekursiv hjælpemetode. Hele klassen er vist nedenfor.

```
public class RekursivList {

	public int max(ArrayList<Integer> list) {
		return maximum(list, 0, list.size() - 1);
	}

	private int maximum(ArrayList<Integer> list, int l, int h) {

		if (l == h) {
			return list.get(l);
		} else {
			int m = (l + h) / 2;
			int max1 = maximum(list, l, m);
			int max2 = maximum(list, m + 1, h);
			if (max1 > max2) {
				return max1;
			} else {
				return max2;
			}
		}
	}
}
```

2. Sortering ved del, løs og kombiner

De to mest effektive sorteringsalgoritmer QuickSort og MergeSort (flettesortering) baserer sig begge på del, løs og kombinér-skabelonen.

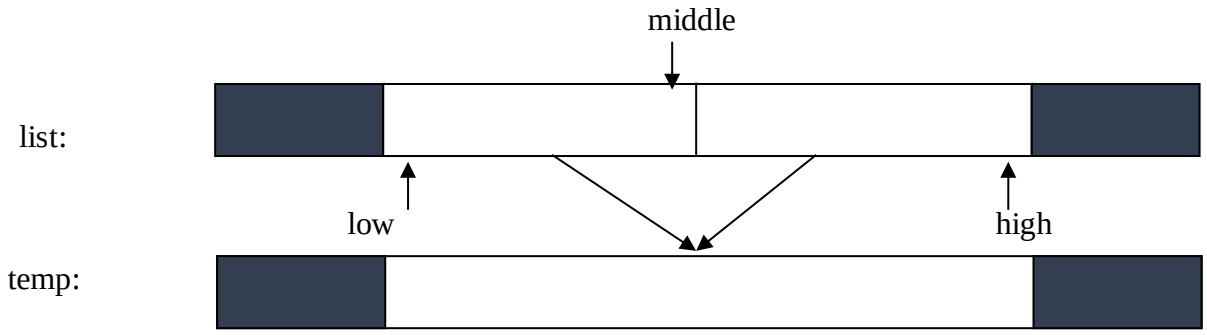
2.1 Flettesortering

Det, der karakteriserer MergeSort er, at her er det let at dele problemet i to, men arbejdet ligger i, at kombinere de to løsninger. Delingen består i at finde indeks for det midterste element i listen, hvorimod kombiner består i at anvende fletteskablelonen på de to sorterede del-lister. Derfor siges MergeSort at være karakteriseret ved: *easy to split – hard to join.*

Flettesortering er den mest naturlige konkretisering af del, løs og kombinér skabelonen, når denne skabelon skal anvendes til at lave en sorteringsalgoritme. Sekvensen der skal sorteres deles i to halvdele, som sorteres hver for sig, hvorefter de flettes sammen til resultatet. Denne ide kan beskrives som følger:

```
private void mergesort(ArrayList<Integer> list, int l, int h) {
      if (l < h) {
            int m = (l + h) / 2;
            mergesort(list, l, m);
            mergesort(list, m + 1, h);
            merge(list, l, m, h);
      }
}
```

Metoden merge skal som antydet foretage en total fletning af `list[low..middle]` og `list[middle+1..high]`. Metoden kan skrives som en konkretisering af fletteskabelonen som følger, hvor der anvendes en lokal variabel `temp` til at flette over i:



Og metodens realisering kan skitseres således:

```
private void merge(ArrayList<Integer> list,
                   int low, int middle, int high) {
     ArrayList<Integer> temp = new ArrayList<>();
     // <flet list[low..middle] og list[middle+1..high] over i
     // en ny liste (temp) ved en total fletning>
     // <list[low..high] = temp>
}
```

Som ved konstruktion af mange andre rekursive metoder, anvendes metoden mergesort fra en public metode der igangsætter rekursionen:

```
public void mergesort(ArrayList<Integer> list) {
     mergesort(list, 0, list.size() - 1);
}
```

Hele klassen får dermed følgende struktur:

```
public class MergeSort {
     private void mergesort(ArrayList<Integer> list, int l, int h) {
          // ...
     }

     private void merge(ArrayList<Integer> list,
                        int low, int middle, int high) {
          // ...
     }

     public void mergesort(ArrayList<Integer> list) {
          // ...
     }
}
```

2.2 Quick sortering

Det, der karakteriserer QuickSort er, at det er svært at dele problemet i to, men let at kombinere de to løsninger. Kombinationsdelen bliver helt tom, hvorimod opdelingen består i at flytte alle elementer mindre end partitions-elementet til venstre for dette og alle elementer til højre. Derfor siges QuickSort at være karakteriseret ved: *hard to split – easy to join.*

Den sorteringsalgoritme der i gennemsnit har den bedste køretid er Quicksort. Quicksort er endvidere et eksempel på anvendelse af del, løs og kombinér skabelonen. Ideen i Quiksort er at opdele den liste der skal sorteres i to dele, hvor elementerne i den ene del er mindre end elementerne i den anden. Ved derefter at sortere de to dele af listen hver for sig, vil hele listen være sorteret

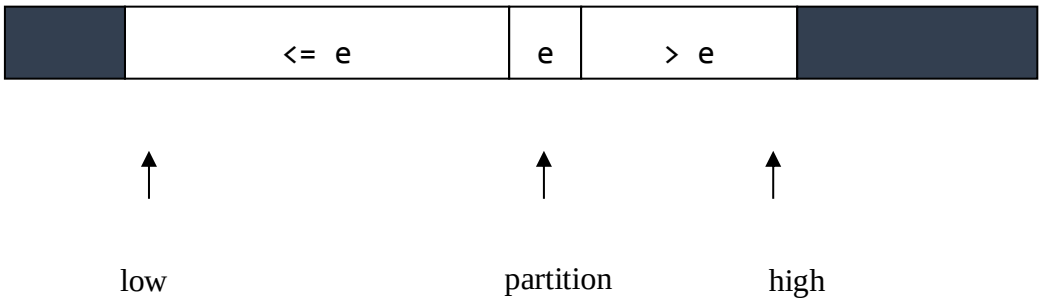
Idéen kan beskrives som følger:

```
private void quicksort(ArrayList<Integer> list, int low, int high) {
      if (low < high) {
            int p = partition(list, low, high);
            quickSort(list, low, p - 1);
            quickSort(list, p + 1, high);
      }
}
```

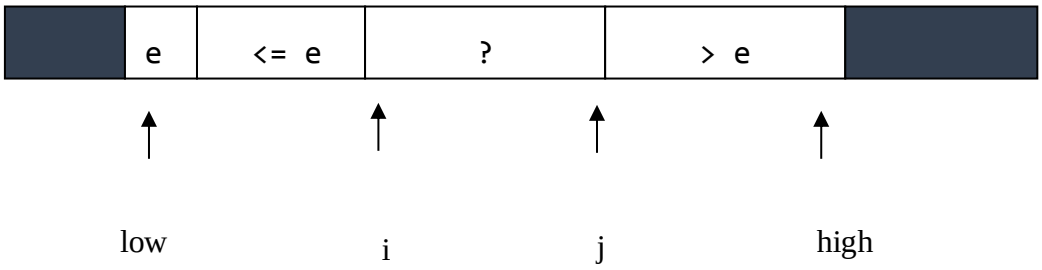
Metoden partition, der foretager opdelingen af `list[low..high]` kan specificeres således:

```
/**
 * Krav: low <= high
 * Returnerer p hvor der i
 * list[low...high] er lavet en ombytning så
 * list[low...p-1] <= list[p] < list[p+1...high]
 */
private int partition(ArrayList<Integer>list, int low, int high) {
     // ...
}
```

Udtrykket i slutbetingelsen kan beskrives ved hjælp af følgende tegning (hvor `e` betegner `list[p]`):



Idéen i realiseringen af Quicksort er, at benytte to indeksvariabler, `i` og `j`, til at afgrænse to dele i `list[low..right]` på følgende måde:



Hvor `e` er såkaldt pivotelement – den værdi vi opdeler i forhold til. Som pivotelement vælger vi `list[low]`.

Hvis `list[i] <= e`, kan den nederste del udvides med `list[i]`, det vil sige `i` kan tælles en op. Tilsvarende kan `j` tælles en ned, hvis `list[j] > e`. Hvis ingen af disse betingelser er opfyldt, betyder det, at `list[i]` hører til i den øverste del, og at `list[j]` hører til i den nederste del; ved at ombytte `list[i]` og `list[j]` kan begge dele udvides. Når `list[i..j]` er tomt, det vil sige når `i > j`, er opdelingen foretaget. Da kan `p` bestemmes, og pivotelementet bringes på plads.

Den endelige realisering bliver hermed:

```
public class QuickSort {
	private void swap(ArrayList<Integer> list, int k, int l) {
		Integer e = list.get(k);
		list.set(k, list.get(l));
		list.set(l, e);
	}

	private void quicksort(ArrayList<Integer> list,int low, int high) {
		if (low < high) {
			int p = partition(list, low,high);
			quickSort(list,low,p-1);
			quickSort(list,p+1,high);
		}
	}

	private int partition(ArrayList<Integer> list, int low, int high) {
		int e = list.get(low);
		int i = low + 1;
		int j = high;
		while (i <=j) {
			if (list.get(i) <= e) { i++; }
			else if (list.get(j) > e) { j--; }
			else {
				swap(list, i, j);
				i++;
				j--;
			}
		}
		swap(list, low, j);
		return j;
	}

	public void quicksort(ArrayList<Integer> list) {
		quicksort(list,0,list.size()-1);
	}
}
```

Det der karakteriserer anvendelsen af del, løs og kombinér i Quicksort er, at opdelingen af et problem i delproblemer er *hårdt* (partition), hvorimod det at kombinere løsningerne er *let* (gør ingenting)