Huvudfrågor

Vilka resurser tar en viss representation och algoritm i anspråk?

Resurs = minnes- eller tidsåtgång

- Rumskomplexitet
- Tidskomplexitet

HUR MÄTER VI DESSA???

Resursåtgång

Minne förbrukas när vi:
- Anropar en metod
- Skapar nya objekt

Minne återfås när vi:
- Avslutar en metod
- Inte längre använder ett objekt (i bästa fall)

***

Tidsåtgången beror i första hand på:
- Antalet anrop till metoder.
- Antalet iterationer.
- (Villkorliga uttryck)

Förbrukad tid är förbrukad tid!!!

Exempel

```
int fact(int n) {

  if( n == 0 )
    return 1;
  else
    return n * fact(n-1);

}
```


Hur beter sig funktioner

Exempel:
- $f(n) = 2n^3 + n^2 + 5n$

n	f(n)	$2n^3$	n^2
10	2150	2000	100
50	252750	250000	2500
100	2010500	2000000	10000
500	250252500	250000000	250000
1000	2001005000	2000000000	1000000

Exempel på komplexitetsklasser

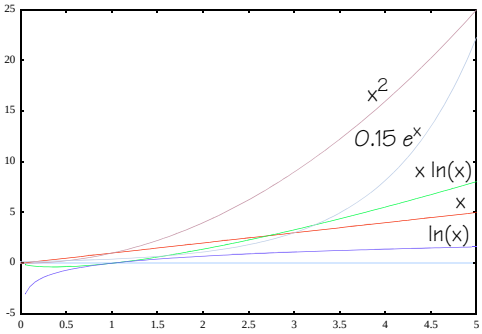
Klass	Benämning
c	konstant
log n	logaritmisk
$\log^2 n$	log-kvadratisk
n	linjär
n log n	n log n
n^2	kvadratisk (polynomisk)
n^3	kubisk (polynomisk)
2^n	exponentiell

Hur lång tid tar det?

n	log(n)	n	n log(n)	n^2	2^n
2	1	2	2	4	4
16	4	16	64	256	$6.5*10^4$
64	6	64	384	4096	$1.84*10^{19}$

OBS: $1.84 * 10^{19}$ mikrosekunder =
$2.14 * 10^8$ dagar =
5 845 århundraden !!!

Komplexitet



Stora-Ordo-notation

En funktion f(n) är Ordo g(n) (skrivs O(g(n))) om det existerar positiva konstanter K och n_0 sådana att:

$$|f(n)| \leq K \cdot |g(n)| \text{ för alla } n \geq n_0$$

Intuitivt:

- Att f(n) är O(g(n)) innebär att f(n) inte växer snabbare är g(n).

Även:

- Att f(n) är Ω(g(n)) innebär att f(n) inte växer långsammare än g(n).
- Att f(n) är Θ(g(n)) innebär att f(n) växer lika snabbt som g(n).

Exempel

```
int fib(int n) {

  if( n == 0 ) return 0;
  else if( n == 1 ) return 1;
  else return fib(n - 1) + fib(n - 2);

}
```


Fibonacci 2

```
int fib1(int n) {
  return fib1(0, 1, n);
}

/*  Om a = fib(m)
    och b = fib(m + 1)
    så är fib1(a, b, n) = fib(m + n) */

int fib1(int a, int b, int n) {
  if( n == 0 ) return a;
  else return fib1(b, a + b, n - 1);
}
```


Fibonacci 3

```
int fib2(int n) {

  int a = 0;
  int b = 1;
  int tmp;

  while( n > 0 ) {
    tmp = a;
    a = b;
    b = b + tmp;
    n--;
  }
  return a;
}
```


Jämför…

```
int power(int x, int n) {

  if( n == 0 )
    return 1;
  else
    return x * power(x, n - 1);

}
```

med…

```
int power(int x, int n) {

  if( n == 0 )
    return 1;
  else if( n % 2 != 0 )        // n is odd
    return x * power(x, n - 1);
  else {                       // n is even
    int m = power(x, n / 2);
    return m * m;
  }

}
```

Binär sökning

```
public int binSearch(int[] a, n) {

  int low  = 0;
  int high = a.length - 1;
  int mid;

  while( low <= high ) {

    mid = (high + low) / 2;

    if( a[mid] < n ) {
      low = mid + 1;
    } else if( a[mid] > n ) {
      hign = mid - 1;
    } else return mid;

  }
  return -1;

}
```


Sammanflätning av arrayer

```
static int[] merge(int[] a, int[] b) {

  int[] c = new int[a.length + b.length];
  int j = 0;

  for( int i = 0; i < a.length; i++ ) {
    while( j < b.length && b[j] < a[i] ) {
      c[i + j] = b[j];
      j++;
    }
    c[i + j] = a[i];
  }
  while( j < b.length ) {
    c[i + j] = b[j];
    j++;
  }
  return c;
}

static void main(String[] args) {

  int[] a = { 1, 4, 6, 8, 9 };
  int[] b = { 3, 5, 7, 8, 10, 12 };

  int[] c = merge(a, b);
  ...
}
```


Sortering 1

```
static void sort(int[] a) {

  int j, tmp;

  for( int i = 0; i < a.length; i++ ) {
    /* Sök efter index till minsta elementet */
    j  = findMin(a, i , a.length);
    tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
  }
}

static int findMin(int[] a, int m, int n) {
  int min = m;

  while( m < n ) {
    if( a[m] < a[min] ) min = m;
    m++;
  }
  return min;
}
```


Sortering 2

```
static void bsort(int[] a) {

  boolean sorted = false;
  int tmp;

  while( ! sorted ) {
    sorted = true;
    for(int i = 1; i < a.length; i++ ) {
      if( a[i] < a[i-1] ) {
        tmp = a[i];
        a[i] = a[i-1];
        a[i-1] = tmp;
        sorted = false;
      }
    }
  }
}
```