F4 1/3 övning på komplexitet

Snabb rep.

Definition:

The complexity of an algorithm is the "cost" to use the algorithm to solve a problem.

Theoretica complexity

4. **The size of the problem**
   (e.g. nbr of inputs).
5. **The nature of input** (sorted/unsorted).
6. **Time complexity of the algorithm itself,** of the method chosen.

We will measure the amount of work the algorithm does in terms of "**elementary operations**".

They are assumed to require one time unit

Complexity is a relative concept, only interesting together with the corresponding elementary operation.

Definition: logarithmic cost criteria

Let T(n) denote the complexity for an algorithm that is applied to a problem of size n.

The size (n in T(n)) of a problem instance (I) is the number of (binary) bits used to represent the instance.

Uniform cost criteria: … you can use **the number of inputs** as problem size since the length of input (in bits) will be a constant times the number of inputs.

Example:

bigProblemToSolve(x_1, x_2, x_3, …, x_n) {…}

Inputs are x_1, x_2, x_3, …, x_n

Then the size of the problem is

$\log(x_1) + \log(x_2) + \log(x_3) + \ldots + \log(x_n)$

this is $\leq n * \log(\max(x_i))$

and $\log(\max(x_i))$ is a constant $\leq$ log(int.max)

so $\leq n * k$

Asymptotic order of growth rate and Big O

Small n is not interesting!

The growth rate of complexity e.g. what happens when n is big, or when we double or

"**Big O**". (Ordo of f, "big O" of f, order of f)

O(f(n)) = {t: .... = t <= c * f(n)

T(n) is O(f(n)) if there exists constants

c>0 and $n_0 \geq 0$ so that for all $n \geq n_0$, we have

$T(n) \leq c^*f(n)$.

We write $T(n) \in O(f(n))$.

We usually use the **worst case** as a measure of complexity.

Sometimes **average** complexity.

And sometimes **amortized** complexity.

Strategy for complexity calculations, quick rep.

1) Set up the requirements, principally which elementary operations (EO) you use
i.e. what cost are you calculating.
(and possibly what you are NOT counting)

2) **Set up the formula for the complexity** rather exactly (mathematically correct) and motivate carefully what you do.

3) **Solve the formula**

thinking about what the result is going to be used for.

Always motivate what you do for instance like this:

(... is the formula that you are working with)

.... = {divide everywhere with 4}

.... = {use formula for geometric sum}

.... = and so on

Recursive functions

```
function fac(n:integer) return integer
   if n <= 1 then
      return 1
   else
      return n*fac(n-1)
   end if
end fac
```

If the call to fac(n) take T(n), then the call to

fac(n-1) should take T(n-1). So we get

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ T(n-1) + c_2 & n > 1 \end{cases}$$

whose solution is O(n).

Solving recursion equations

means that we try to express them in closed form, i.e without T(...) terms on the right hand side.

Mergesort is a famous sorting algorithm

```
vektor mergesort (v:vektor; n:integer)
   // v is an array of length n
   if n = 1 then
      return v
   else
      split v in two halfs v1 och v2,
                        with length n/2
      return merge(mergesort(v1, n/2),
                   mergesort(v2, n/2))
   end if
end mergesort
```

Merge takes two sorted arrays and merges these with one another to a sorted list. If T(n) is the wc-time and we assume n is a power of 2, we get:

$$T(n) = \begin{cases} c_1 & if\, n = 1 \\ 2T(n/2) + c_2 n & if\, n > 1 \end{cases}$$

The first "2" is the number of subsolutions and n/2 is the size of the subsolutions

$c_2 n$ is the test to discover that n!=1, ($O(1)$), to break the list in two parts ($O(1)$) and to merge them ($O(n)$). The solution is $O(n \log n)$

Problem 1. Analysera Selectionsort

```
st void swap(int[] f), int x, int y {
   int tmp = f[x];
   f[x] = f[y];
   f[y] = tmp;
}
1.  st void selectionSort(int[] f) {
2.     int lowIndex = 0;
3.     for (int slot2fill = 0;
4.             slot2fill < f.length-1;
5.                slot2fill++) {
       //slot2fill står i tur att ordnas
6.        lowIndex = slot2fill; // minst
7.        for (int j = slot2fill+1;
8.                   j < f.length;
9.                              j++) {
10.          if (f[j]<f[lowIndex]){
11.             lowIndex = j;
              }
           }
12.        if (lowIndex != slot2fill) {
13.           swap(lowIndex,slot2fill, f);
           }
       }
   }
```

Problem 2

Lös

$T(1) = 2$

$T(n) = T(n-1) + 2$

Problem 3

Lös

$T(n) = \{\text{if } n=1 \text{ then } c_1 \text{ else } 2T(n/2) + c_2 n\}$

Problem 4

Suppose you have algorithms with the six running times listed below. (Assume these are the exact running times.)

How much slower do each of these algorithms get when you

(a) increase the input size by one

or

(b) double the input size?

(i) n^2

(ii) n^3

(iii) $100n^2$

(iv) logn

(v) n log n

(vi) 2^n

Problem 5

Logaritm lagar man bör kunna

(alla är inte individuella lagar)

$$y = a^x \Rightarrow x = {}^a\log y = \frac{{}^b\log y}{{}^b\log a} = {}^a\log b \cdot {}^b\log y$$

$$a^{{}^b\log n} = n^{{}^b\log a}$$

$${}^a\log a = 1, \quad {}^a\log 1 = 0$$

$$\log a^x = x \cdot \log a \quad \text{in particular} \quad {}^a\log a^x = x$$

$$\log (x \cdot y) = \log x + \log y \qquad \log \frac{x}{y} = \log x - \log y$$

$$a^x \cdot a^y = a^{x+y} \qquad a^x \cdot b^x = (a \cdot b)^x$$

$$(a^x)^y = a^{x \cdot y} \quad \text{in particular} \quad (2^2)^i = 2^{2i} = (2^i)^2$$

Problem 6

Några komplexitetsfunktioner som är bra att kunna:

$T(n/2) + c \quad \in O(\log n)$

$T(n/2) + c\log n \quad \in O(\log n)^2$

$T(n/2) + cn \quad \in O(n)$

$T(n/2) + n^2 \quad \in O(n^2)$

$2T(n/2) + c \quad \in O(n)$

$2T(n/2) + c\log n \quad \in O(n)$

$2T(n/2) + cn \quad \in O(n\log n)$

$2T(n/2) + cn\log n \in O(n(\log n)^2)$

$T(n-1) + c \quad \in O(n)$

$T(n-1) + c\log n \quad \in O(n\log n)$

$T(n-1) + cn \quad \in O(n^2)$

$T(n-1) + cn\log n \in O(n^2\log n)$

$2T(n-1) + c \quad \in O(2^n)$

Problem 7 Summor man bör kunna

a) $\sum_{i=1}^{n} 1 = n$

b) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

c) $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \in O(n^3)$

d) $\sum_{i=1}^{n} i^k \in O(n^{k+1})$

e) $\sum_{i=0}^{n} a \cdot x^i = a \cdot \frac{x^{n-1} - 1}{x - 1}, \; x \,!= 1$

f) $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

g) $\sum_{i=0}^{n} i \cdot x^i \in O(nx^{n+2})$

h) $\sum_{i=0}^{n} i \cdot 2^i \in O(n \cdot 2^{n+2})$

i) $\sum_{i=1}^{n} \frac{1}{i} \approx ln n$

j) $\sum_{i=2}^{n} logi \in O(nlogn)$

k) $\sum_{i=2}^{n} i \cdot logi \in O(n^2 \, logn)$

l) $\sum_{i=k}^{p} 1 = p - k + 1$

$\sum_i c \cdot i = c \cdot \sum_i i$	$\sum_{i=c}^{n} i = \sum_{i=0}^{n-c} i + c$	$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$	$\sum_{i=0}^{n} (n - i) = \sum_{i=0}^{n} i$