Calculating the sum of all elements in a list:

### sum/1

```haskell```
def sum([]) do 0 end
```
def sum([h|t]) do
  s = sum(t)
  h + s
end```

### sum/2

```haskell```
def sum([], s) do s end
```
def sum([h|t], s) do
  s1 = h+s
  sum(t, s1)
end```

What are the run-time complexities of sum/1 and sum/2?

---

1. **run-time complexity of foo**

### foo/1

```haskell```
def foo([]) do [] end
def foo([h|t]) do
  z = foo(t)
  bar(z, [h])
end```

### foo/2

```haskell```
def foo([], y) do y end
def foo([h|t], y) do
  z = zot(h, y)
  foo(t, z)
end```

What are the run-time complexities of foo/1 and foo/2?

---

2. **run-time complexity of reverse**

### nreverse/1

```haskell```
def nreverse([]) do [] end
def nreverse([h|t]) do
  z = nreverse(t)
  append(z, [h])
end```

### reverse/2

```haskell```
def reverse([], y) do y end
def reverse([h|t], y) do
  z = cons(h, y)
  reverse(t, z)
end```

What are the run-time complexities of nreverse/1 and reverse/2?
Assume that \( \text{append/2} \) takes \( kn \) ms to execute, where \( k \) is some constant time and \( n \) is the length of the list.

Describe the time \( T_n \) it takes to execute \( \text{reverse/1} \) of a list of length \( n \):

\[
T_0 = a \text{ ms}
\]

\[
T_n = T_{n-1} + k(n-1) + b \text{ ms}
\]

\[
T_n = T_{n-1} + k(n-1) + b
\]

\[
= T_{n-2} + k(n-2) + k(n-1) + 2b
\]

\[
= T_{n-3} + k(n-3) + k(n-2) + k(n-1) + 2b
\]

\[
= \ldots + (n-1)k + nb
\]

\[
= \ldots + (n-1)k + nb
\]

\[
= (\frac{k}{2})n^2 - \frac{k}{2}n + bn + a
\]

\[
T_n \in O(n^2)
\]

We know:

\[
T_n = \frac{k}{2}n^2 - \frac{k}{2}n + bn + a
\]
run-time complexity of reverse/1

```
def nreverse([]) do end
    def nreverse([h|t]) do
        z = nreverse(t)
        append(z, [h])
    end
```

run-time complexity of reverse/2

```
def reverse([], y) do y end
    def reverse([h|t], y) do
        z = cons(h, y)
        reverse(t, z)
    end
```

complexity of quick-sort

```
def qsort([]) do [] end
def qsort([h]) do [h] end
def qsort(all) do
    {low, high} = partition(all)
    lowS = qsort(low)
    highS = qsort(high)
    append(lowS, highS)
end
```

the recurrence relation

\[
T_1 = a \\
T_n = 2 \times T_{n/2} + nc \\
\]

\[
= 2 \times (2 \times T_{n/4} + (n/2)c) + nc \\
= 4 \times T_{n/4} + 2 \times nc \\
= 8 \times T_{n/8} + 3 \times nc \\
\vdots \\
= 2^k \times T_1 + k \times nc \\
= 2^\log_2(n) \times a + \log(n) \times nc \\
= n \times a + \log(n)n \times c
\]
### Complexity of Quick-Sort

-Qsort worst case

What if we run qsort on an already ordered list?

### Complexity of Merge-Sort

-What is done in each iteration?
-How many iterations do we have?
-What is the run-time complexity?
-Which is best qsort or msort?

```euphoria
def msort([]) do [] end
def msort(l) do
{a, b} = split(l)
as = msort(a)
bs = msort(b)
merge(as, bs)
end
```

### Complexity of Fibonacci

-What is done in each iteration?
-How many iterations do we have?

```euphoria
def fib(0) do 0 end
def fib(1) do 1 end
def fib(n) do
  fib(n-1) + fib(n-2)
end
```
the recurrence relation

Let’s cheat a bit to make it simpler:

\[
T_0 = a
\]

\[
T_n = 2 \times T_{n-1} + c
\]

\[
= 2 \times (2 \times T_{n-2} + c) + c
\]

\[
= 4 \times T_{n-2} + 3 \times c
\]

\[
= 8 \times T_{n-3} + 7 \times c
\]

\[
\vdots
\]

\[
= 2^n \times T_0 + (2^n - 1) \times c
\]

\[
= 2^n \times a + 2^n \times c - c
\]

The more precise answer is \(O(1.6^n)\)

complexity of fibonacci

\[
\text{fibonacci/1}
\]

\[
\begin{array}{c|c|c|c|c|c}
\text{n} & \text{2^n} \\
\hline
\text{\textbullet} & \text{\textbullet} & \text{\textbullet} & \text{\textbullet} & \text{\textbullet} & \text{\textbullet}
\end{array}
\]

The smarter implementation is \(O(n)\)

... an even smart solution is \(O(\log(n))\)

The big question

What is the difference between a smart programmer and a not so smart programmer?

3 billion years?

operations on trees

Let’s represent trees as:

\[
\text{:nil}
\]

{:node, key, value, left, right}

- new: create a empty tree
- insert: add an element to the three
- lookup: search for an element
- modify: modify an element
why trees?

Why use trees, why not use lists?

benchmark tree operations

Operations on a tree.

Figure: Execution time in ms of 100.000 calls

why trees?

tuples as a key value store

def new([a,b,c]) do {a,b,c} end

def lookup({a,_,_}, 1) do a end

def lookup({_, b,_}, 2) do b end
:
def modify({_,b,c}, 1, v) do {v, b, c} end

def modify({a,_,c}, 2, v) do {a, v, c} end
:

why trees?

Why use trees, why not use tuples?
tuples using built-in functions

```erlang
def new(list) do List.to_tuple(list) end
def lookup(tup, k) do elem(tup, k) end
def modify(tup, k, v) do put_elem(tup, k, v) end
```

*The functions put_elem/3 will create a copy of the original tuple!*

benchmark tuple operations

Operations on a tuple.

```
```

```
```

Figure: Execution time in ms of 100,000 calls

compare tuples and trees

Tuple vs tree.

```
```

```
```

Figure: Modify operations, execution time in ms of 100,000 calls

root of all evil

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97 percent of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3 percent.

*Donald Knuth*
code vs time

- code size
- execution time

programming rules

- understand the problem before starting coding
- write well structured code that is easy to understand
- use abstractions to separate functionality from implementation
- think about complexity
- benchmark your program
- if needed, optimize