Math 124 - Programming for Mathematical Applications

UC Berkeley, Spring 2023

Homework 7

Due Wednesday, March 15

Problem 1

Consider a floating point representation with precision $p = 2$:

$$+(d_0 + d_1 2^{-1})2^e, \qquad 0 \leq d_i < 2$$

normalized so that $d_0 \neq 0$ and with the exponent e ranging from $e_{\min} = -2$ to $e_{\max} = 2$.

Problem 1(a)

List the numbers that can be represented in this format.

Observe that the $d_1 2^{-1}$ term can be $0, 2^{-1}$. The 2^e term can be $2^{-2}, 2^{-1}, 2^0, 2^1, 2^2$. So we have two terms that the stuff inside the parenthesis can be evaluated to: $1, 1.5$. Then multiply each by the values of 2^e to get

$$1 \mapsto \left\{ \frac{1}{4}, \frac{1}{2}, 1, 2, 4 \right\}$$
$$\frac{3}{2} \mapsto \left\{ \frac{3}{8}, \frac{3}{4}, \frac{3}{2}, 3, 6 \right\}$$

Problem 1(b)

If the precision is increased from $p = 2$ to $q > p$, how many new numbers would there be between two previous adjacent numbers (not including the endpoints)?

For each time we increase p by 1, we divide each interval between the two numbers by 2 into two new intervals. So there will be 2^{q-p} new intervals. The number of numbers in the number of intervals minus 1. So there are $2^{q-p}-1$ new numbers.

Problem 1(c)

Between an adjacent pair of nonzero `Float32` floating point numbers, how many `Float64` numbers are there?

The precision in Float32 is 24 while the precision in Float64 is 53. So between two adjacent non-zero numbers in Float32, there are $2^{29}-1$ Float64 numbers. If the sign bit is taken into account, the difference is still the same: 52 - 23 = 29

Problem 1(d)

The floating point numbers include many integers, but not all of them. What is the smallest postive integer n that is not exactly represented as a `Float64` ? *Hint*: First consider $p = 1, 2, \ldots$, and try to see the pattern.

The number of bits for the exponent is 11. Since the precision exceeds the number of bits of the exponent, any increase in precision does not change the largest positive integer represented. So the smallest integer not represented is $2^{2^{10}-1}+1$

Problem 2

Find the asymptotic operation counts and memory usage for the Julia functions below, using Big O notation.

Problem 2(a)

In [1]:
```
function p2a(n)
    s = 0
    for i = 1:n
        if i % 3 == 0
            s += i^2
        end
    end
    s
end
```

Out[1]:
```
p2a (generic function with 1 method)
```

The mod operation, comparison, addition, and squaring are looped over n times, which gives an operation count of $\mathcal{O}(n)$. The storage is only setting variables to a number, so it is constant : $\mathcal{O}(1)$.

Problem 2(b)

In [2]:
```
function p2b(n)
    digits = Int64[]
    while n > 0
        push!(digits, n % 2)
        n = n ÷ 2
    end
    digits
end
```

Out[2]:
```
p2b (generic function with 1 method)
```

The operations mod, push, and integer division are performed n, n, and $\frac{n}{2}$ times. So the operation count is $\mathcal{O}(n)$. The storage used is a vector of size n. So the memory is also $\mathcal{O}(n)$.

Problem 2(c)

In [3]:
```
function p2c(x)
    n = length(x)
    alldiff = [ abs(x[i] - x[j]) for i = 1:n, j = 1:n ]
    for i = 1:n
        alldiff[i,i] += Inf
    end
    mindiff = minimum(alldiff)
end
```

Out[3]:
```
p2c (generic function with 1 method)
```

The operations used are subtraction, absolute value, addition, and minimum. Each of these are performed about n^2 times with the exception of addition with is n. So operation count is $\mathcal{O}(n^2)$. The storage is an $n \times n$ matrix, so memory is also $\mathcal{O}(n^2)$.

Problem 2(d)

Write a function `p2d(x)` which computes the same thing as `p2c(x)` but using asymptotically less operations and memory, and find its asymptotical operations and memory usage as before.

In [28]:
```
function p2d(x)
    function merge(L, R, x)
        i = j = k = 1
        while i ≤ length(L) && j ≤ length(R)
            if L[i] < R[j]
                x[k] = L[i]
                i += 1
            else
                x[k] = R[j]
                j += 1
            end
            k += 1
        end

        while i <= length(L)
            x[k] = L[i]
            i += 1
            k += 1
        end

        while j <= length(R)
            x[k] = R[j]
            j += 1
            k += 1
        end
    end
```

```
    end

    function merge_sort(x)
        if length(x) == 1
            x
        else
            mid = length(x) ÷ 2
            L = x[1:mid]
            R = x[mid + 1:end]

            merge_sort(L)
            merge_sort(R)

            merge(L, R, x)
        end
    end

    merge_sort(x)
    min = Inf
    for i = 1:(length(x) - 1)
        diff = abs(x[i] - x[i + 1])
        if diff < min
            min = diff
        end
    end
    min
end
```

Out[28]: p2d (generic function with 1 method)

The function uses merge sort to sort the numbers. As it moves from left to right along the number line, it saves the least distance it has seen. Merge sort has an operation count of $n\log(n)$. Finding the minimum has a count of n since it loops through each operation $n-1$ times. The memory is only the length of the vector that we are trying to find the minimum difference of. So operation count is $\mathcal{O}(n\log(n))$ and memory is $\mathcal{O}(n)$.