Chapter_7_Iteration

March 6, 2024

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the following formula: $y = (x + a/x)/2$

[1]:
```
# Example
a = 4
x = 3
y = (x + a/x) / 2
y
```

[1]:
```
2.1666666666666665
```

The result is closer to the correct answer ($\sqrt{4} = 2$). If we repeat the process with the new estimate, it gets even closer:

[2]:
```
x = y
y = (x + a/x) / 2
#After a few more updates, the estimate is almost exact:
x = y
y = (x + a/x) / 2
x = y
y = (x + a/x) / 2
y
```

[2]:
```
2.0000000000262146
```

In general we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

When y == x, we can stop. Here is a loop that starts with an initial estimate, x, and improves it until it stops changing

[3]:
```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

```
2.0000102400262145
2.0000000000262146
2.0
```

For most values of a this works fine, but in general it is dangerous to test float equality. Floating-point values are only approximately right: most rational numbers, like 1/3, and irrational numbers, like $\sqrt{2}$, can't be represented exactly with a float.

Rather than checking whether x and y are exactly equal, it is safer to use the built-in function abs to compute the absolute value, or magnitude, of the difference between them:

[4]:
```
# if abs(y-x) < epsilon:
#     break
```

Where epsilon has a value like 0.0000001 that determines how close is close enough.

Executing algorithms is boring, but designing them is interesting, intellectually challenging, and a central part of computer science.

[ ]:
```

```