Rapport:

Kommentarer hetet fra koden om tidsbruket:

Om metode 1:

"//Denne metoden vil skalere lineært med 'n', altså den vil ha like mange metode kall som verdien av 'n'
//dette fordi den vil kalle f. eks. P(5, x) -> P(4, x) -> P(3, x) -> P(2, x) -> P(1, x),
//dette siden den kaller seg selv med 'n - 1' hver gang helt ned til 'n == 0', hvor den da returnerer 1"

Om metode 2:

"//Denne metoden vil skalere med log(base 2)(n) + 1, si f.eks. n = 32 så vil den kalle P(32, x) -> P(16, x) -> P(8, x) -> P(4, x) -> P(2, x) -> P(1, x) altså 6 metode kall og log(base 2)(32) + 1 = 5(avrundert) + 1 = 6
//dette gjelder også når metoden får en 'n'-verdi som er et oddetall da den "trekker" en 'x' ut av utrykket for å få 'n - 1' for å få et partall som igjen kan halveres
//ta f. eks. n = 35 så vil metoden kalle P(35, x) -> x * P(17, x) -> x * P(8, x) -> P(4, x) -> P(2, x) -> P(1, x) altså 6 metode kall og log(base 2)(35) + 1 = 5(avrundet) + 1 = 6
//dette gjør at metoden må ha veldig store 'n' verdier for at den skal bli treger
//sammmenlignet med potens1-metoden hvis 'n' er si 5000 så vil potens1 ha 5000 rekusive kall, mens denne vil ha estimert log(base 2)(5000) + 1 = 12(avrundet) + 1 = 13"

"Analyse":

"Den andre rekursive metoden er som raskest i de tilfellene jeg har sett på. Dette kan komme av at den første bare går gjennom som en løkke og ganger 'x'-verdien med seg selv 'n'-antall ganger, ergo O(n)
derimot i den andre rekursive metoden så er vi mer effektive å sier at hvis 'n' er et partall så vil x^n = (x*x)^(n/2) , her vil 'n' bli halvert så si 'n = 500' metode 1 vil på et kall ha kommet til P(499, x) mens den andre vil ha kommet til P(250, x), basert på forklaringene mine om metode to kan vi også si at metode 2 har O(log(base 2)(n)), videre kan vi også trekke at metode 2 vil skalere saktere med 'n' enn det metode 1 vil"

Alle testresultatene er kjørt på 1.000.000 gjennomganger, da dette var et tall som ga mer determentiske resultater var det mye mindre var det ofte store forskjeller mellom hver kjøring på samme metode, ved 1.000.000 gjennomganger fikk jeg et avvik på noen få nanosekunder bare, derfor valgte jeg å fortsette med dette da dette ville gi best sammenlingbarhet mellom de forskjellige verdiene som tests ut. Jeg kjørte også Java sin Math.Pow() etter første og andre metode og så at jeg selv mellom disse fikk avvik på noen nanosekunder hvis jeg hadde 100.00 gjennomganger bare, mens ved 1.000.000 gjennomganger så jeg aldri en forskjell på mer en 1 nanosekund.

Testresultater

n = 50 x = 1.001

Metode 1: 119 nanosekunder | verdi: 1.051244832434745

Metode 2: 41 nanosekunder | verdi: 1.0512448324347439

Java(Math.Pow()): 55 nanosekunder | verdi: 1.0512448324347454

---

n = 5000 x = 1.001

Metode 1: 13580 nanosekunder | verdi: 148.04283616255938

Metode 2: 51 nanosekunder | verdi: 148.04283616253903

Java(Math.Pow()): 68 nanosekunder | verdi: 148.0428361625591

---

n = 50 x = 1.023

Metode 1: 121 nanosekunder | verdi: 3.1173221513939593

Metode 2: 44 nanosekunder | verdi: 3.117322151393958

Java(Math.Pow()): 54 nanosekunder | verdi: 3.1173221513939597

---

n = 5000 x = 1.023

Metode 1: 13212 nanosekunder | verdi: 2.3887382334432986E49

Metode 2: 47 nanosekunder | verdi: 2.3887382334431864E49

Java(Math.Pow()): 64 nanosekunder | verdi: 2.3887382334432986E49

---

n =10 x = 2

Metode 1: 47 nanosekunder | verdi: 1024

Metode 2: 40 nanosekunder | verdi: 1024

Java(Math.Pow()): 55 nanosekunder | verdi: 1024

---

n = 14 x = 3

Metode 1: 53 nanosekunder | verdi: 4782969

Metode 2: 38 nanosekunder | verdi: 4782969

Java(Math.Pow()): 54 nanosekunder | verdi: 4782969

Konklusjon:

Som antat i analysen av metodene så er metode 2 alltid raskere enn metode 1, men for verdier av 'n' som er lave utgjør det ikke så stor forskjell, men som vi så i 'n = 5000' tilfellene så blir metode 1 helt ubrukelig treg i forhold. Det som er mer overraskende er at metode 2 også var raskere en Java sin Math.pow()-metode i alle tilfellene, men forskjellen vokste ikke med 'n'. Det kan hende at metoden har noen ekstra sikkerhets sjekker eller tolkning av input'en for å kjøre optimalt, noe som koster den noen få nanosekunder.

Når 'n' ble veldig høy fikk vi litt variasjon i svarende, men dette skyldes nok heller avrundingsfeil grunnet størrelsen på tallet, og feilen kommer ikke før i 10. desimaltall så feilmarginene er ikke så stor, men en interessant oppservasjon var at metode 1 og Math.pow()-metoden stemmte, i alle tilfellene jeg har sett, mer enn det metode 2 gjorde med dem. Dette har nok med hvordan utregningen blir kalt (rekkefølge). For å se litt mer på det så skrev jeg om returnen i metode 2 fra å ta (x^2)^(n/2) til (x^(n/2))^2 som er matematisk likt, men ga forskjellig resultater. Det viste seg i mitt tilfelle at (x^2)^(n/2) ga et svar nærmere det metode 1 og Math.pow() ga.