Cost of Algorithms

Inputs: parameterized by an integer n, called the size

$T(I)$ = runtimes on input I → runtime of a particular instance

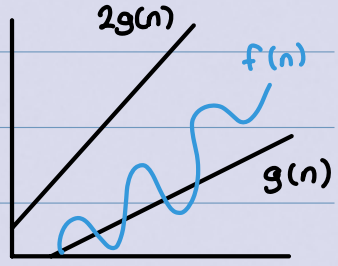
$T_{worst}(n) = \max_{I \text{ of size } n}(T(I))$ → worst-case runtime (default)

$T_{best}(n) = \min_{I \text{ of size } n}(T(I))$ → best-case runtime, not used much

Asymptotic Notation

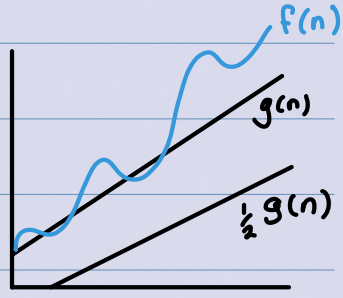
- consider 2 functions $f(n)$ and $g(n)$ with values in $\mathbb{R}_{>0}$

big-O: we say that $f(n) \in O(g(n))$ if there exists a $C > 0$ and n_0 such that for $n \geq n_0$, $f(n) \leq C\,g(n)$



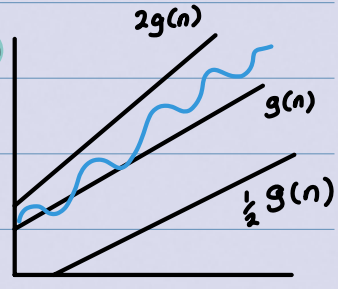
big-Ω: we say that $f(n) \in \Omega(g(n))$ if there exists a $C > 0$ and n_0 such that for $n \geq n_0$, $f(n) \geq C\,g(n)$

⇒ equivalent to $g(n) \in O(f(n))$



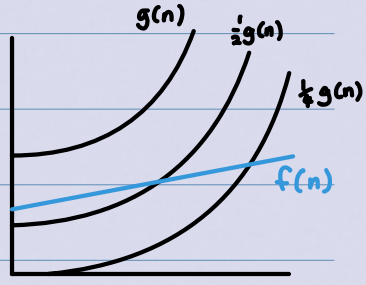
Θ: we say that $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

⇒ in particular true if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C$ for some $0 < C < \infty$



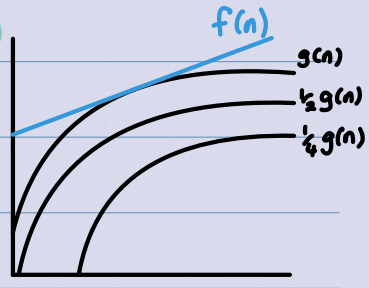
little-o: we say that $f(n) \in o(g(n))$ if for all $C > 0$, there exists an n_0 such that for $n \geq n_0$, $f(n) \leq C\,g(n)$

⇒ equivalent to $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$



little-ω: we say that $f(n) \in \omega(g(n))$ if for all $c > 0$, there exists an n_0 such that for $n \geq n_0$, $f(n) \geq cg(n)$

⇒ equivalent to $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$



Examples

a) $n^k + c_{k-1}n^{k-1} + \ldots + c_0 \in \Theta(n^k)$

b) $2^{n-1} \notin \Theta(2^n)$

c) $(n-1)! \in \Theta(n!)$

Definitions for Several Parameters

consider two functions $f(n,m)$, $g(n,m)$ with values in $\mathbb{R}_{>0}$

$f(n,m)$ is in $O(g(n,m))$ if there exist c, n_0, m_0 such that $f(n,m) \leq Cg(n,m)$ for $n \geq n_0$ or $m \geq m_0$

Case Study: Maximum Subarray

Given an array $A[0,\ldots,n]$, find a contiguos subarray $A[i,\ldots,j]$ that maximises the sum $A[i] + \ldots + A[j]$.

Example: given $A = [10, -5, 4, 3, -5, 6, -1, -1]$, the subarray $A[0,\ldots,5] = [10, -5, 4, 3, -5, 6]$ has sum $10-5+4+3-5+6=13$ is the max.

Brute Force algorithm:

runtime is $\Theta(n^3)$

but, we can improve on this!
idea: we recompute the same sum many times in the j loop

```
BruteForce (A)
1. opt = 0
2. for (i=0→n) {
3.     for (j=i→n) {
4.         sum = 0
5.         for (k=i→j) {
6.             sum += A[k]
```

```
BetterBruteForce(A)
1. opt = 0
2. for (i=0→n) {
3.   sum = 0
4.   for (j=1→n) {
5.     sum += A[j]
6.     if (sum > opt) {
7.       opt = sum
8.     }
9.   }
10. }
11. return opt
```

```
7.     }
8.     if (sum > opt) {
9.       opt = sum
10.    }
11.  }
12. }
13. return opt
```

⇒ runtime $O(n^2)$

but, we can still do better using a **divide-and-conquer approach**:

idea: solve the problem twice in size $n/2$ (assuming n is a power of 2). Then, the **optimal subarray** (if not empty):

1. is completely **in the left half** $A[0, \dots, n/2]$
2. or is completely **in the right half** $A[n/2+1, \dots, n]$
3. or **contains both** $A[n/2]$ **and** $A[n/2+1]$

⇒ the three cases are mutually exclusive

to find the optimal subarray in **case 3**, we have:

$$A[i] + \dots + A[n/2] + A[n/2+1] + \dots + A[j]$$

more abstractly, we have $F(i,j) = f(i) + g(j)$, for $i \in [0, \dots, n/2]$ and $j \in [n/2+1, \dots, n]$.

To maximise $F(i,j)$, we **maximise $f(i)$ and $g(j)$ independently!**

```
MaximiseLowerHalf(A)
1. opt = A[n/2]
```

```
MaximiseUpperHalf(A)
1. opt = A[n/2+1]
```

```
2. sum = A[n/2]
3. for (i = n/2 - 1 → 0) {
4.     sum += A[i]
5.     if (sum > opt) {
6.         opt = sum
7.     }
8. }
9. return opt
```

```
2. sum = A[n/2 + 1]
3. for (i = n/2 + 1 → n) {
4.     sum += A[i]
5.     if (sum > opt) {
6.         opt = sum
7.     }
8. }
9. return opt
```

↳ runtimes are $\Theta(n)$! ↵

So, final divide and conquer algorithm:

```
DivideAndConquerMaximumSubarray(A[0, ..., n])
1. if (n == 1) return max(A[0], 0)
2. opt_low = DivideAndConquerMaximumSubarray(A[0, ..., n/2])
3. opt_high = DivideAndConquerMaximumSubarray(A[n/2+1, ..., n])
4. opt_mid = MaximiseLowerHalf(A) + MaximiseUpperHalf(A)
5. return max(opt_low, opt_mid, opt_high)
```

⇒ runtime: $T(n) = 2T(n/2) + \Theta(n) \in \Theta(n \log n)$