Smallest Simple Function for Big Oh

- If $f(n)$ is O(n^2), is it also O(n^3)?
  - Since O(n^3) grows faster than O(n^2), it is true.
  - However, O(n^3) over estimates an O(n^2) function.
- So, our attempt will be to find the smallest simple function for which $f(n)$ is O($g(n)$).
- Some well known growth functions in order of growth:
  - 1, $\log n$, n, $n \log n$, n^2, n^3, 2^n, etc.
- Notice that only +ve integral values of n are of interest.

Guidelines for Computing Big Oh

- Find the dominant term of the function and find its order.
  - A logarithmic function dominates all constants.
  - A polynomial function dominates all logarithmic functions.
  - A polynomial of degree k dominates all lower degree polynomials.
  - An exponential function dominates all polynomial functions.
- Basis here is that:
  - The dominant term grows more rapidly compared to others.
  - It will quickly outgrow non-dominant terms.

Other Simple Rules

- If $\mathrm{T}_1(n) = \mathrm{O}(f_1(n))$, and $\mathrm{T}_2(n) = \mathrm{O}(f_2(n))$, then
  - $\mathrm{T}_1(n) + \mathrm{T}_2(n) = \max\{\mathrm{O}(f_1(n)), \mathrm{O}(f_2(n))\}$
  - $\mathrm{T}_1(n) * \mathrm{T}_2(n) = \mathrm{O}(f_1(n) * f_2(n))$
- If $\mathrm{T}(n)$ is a polynomial of k then $\mathrm{T}(n) = \Theta(n^k)$[1]
- $\log^k n = O(n)$ for any constant k
- For checking whether $g(n)$ and $f(n)$ are comparable find $\lim \frac{f(n)}{g(n)} \leq k$, where $k > 0$ is a constant?
- E.g.: $\lim \frac{n^2}{n^2+6} = \lim \frac{2n}{2n} = 1$
- $\lim \frac{\log n}{\log n^2} = \lim \frac{(1/n)}{2(1/n)} = 1/2$.

[1] Not defined yet

Some Examples

- Examples of O(n^2) functions: n^2, n^2+n, $n^2+1000n$, 100n^2+ 1000n, n, $n/100$, $n^{1.99999}$, $n^2/(\log\log\log n)$
- $\log n! = O(n\log n)$:

$$\begin{aligned}\log n! &= \log 1 + \log 2 + \ldots + \log n \\ &\leq \log n + \log n + \ldots + \log n = n\log n\end{aligned}$$

- $2^{n+1} = 2.2^n$ for all n.
  - So with $c = 2, n_0 = 1$, $2^{n+1} = O(2^n)$.
- But $2^{2n} \neq O(2^n)$ can be proved by contradiction.
  - We have $0 \leq 2^{2n} = 2^n.2^n \leq c.2^n$, then $2^n \leq c$.
  - But no constant is greater than 2^n.

Some Proofs for Big Oh

Exercise 1

Prove that $n^3 + 20n + 1$ is not O(n^2).

Solution

- Assume that $n^3 + 20n + 1$ is O(n^2).
- By definition of big-Oh it implies $n^3 + 20n + 1 \leq c.n^2$.
- Divide both side of the inequality by n^2.
- So, $n + \frac{20}{n} + \frac{1}{n} \leq c$.
- Since left side grows with n, c cannot be a constant.

Some Proofs for Big Oh

Exercise 2

Prove that $f(n) = \frac{n^2+5\log n}{2n+1}$ is O(n)

Solution

- $5\log n < 5n < 5n^2$, for all $n > 1$
- $2n + 1 > 2n$, so $\frac{1}{2n+1} < \frac{1}{2n}$ for all $n > 0$
- Thus $\frac{n^2+5\log n}{2n+1} \leq \frac{n^2+5n^2}{2n} = 3n$ for all $n > 1$.
- So, with $c = 3$ and $n_0 = 1$ we have $f(n) < c.n$

Some Proofs for Big Oh

Exercise 3

Let $f(n) = n^k$, and $m > k$, then $f(n) = O(n^{m-\epsilon})$, where $\epsilon > 0$

Solution

- Set $\epsilon = (m-k)/2$, so $m - \epsilon = (m+k)/2 > k$.
- Hence, $n^{(m-\epsilon)}$ dominates n^k.

Some Proofs for Big Oh

Exercise 4

Let $f(n) = n^k$, and $m < k$, then $f(n) = \Omega(n^{m+\epsilon})$[a], where $\epsilon > 0$

[a] Ω not defined yet

Solution

- Set $\epsilon = (k - m)/2$, so $m + \epsilon = (m + k)/2 < k$.
- Hence, $n^{(m+\epsilon)}$ is dominated by n^k.

Some Proofs for Big Oh

Exercise 5

Show $f(n) = n^k$ is of O($n^{\log\log n}$) for any constant $k > 0$

Solution

- $n^k < n^{\log\log n}$ iff $k < \log\log n$, i.e., $n > 2^{2^k}$.
- Setting $n_0 = 2^{2^k}$, we have $n^k = O(n^{\log\log n})$.

Computing Big Oh of Programs

- Single loops: **for**, **while**, **do-while**, **repeat until**
  - Number of operations is equal to number of iterations times the operations in each statement inside loop.
- Nested loops:
  - Number of statements in all loops times the product of the loop sizes.
- Consecutive statements:
  - Use addition rule: O($f(n)$) + O($g(n)$) = max($g(n), f(n)$)
- Conditional statement:
  - Number of operations is equal to running time of conditional evaluation and the maximum of running time of **if** and **else** clauses.

Computing Big Oh of Programs

- **Switch** statements:
  - Take the complexity of the most expensive case (with the highest number of operations).
- Function calls:
  - First, evaluate the complexity of the method being called.
- Recursive calls:
  - Write down recurrence relation of running time.
  - Solution mostly possible by observing pattern of growth and prove the same on the basis of induction from the base case.
  - For divide and conquer algorithms Master Theorem can be used.

Analysis of for Loops

```
for (i = 0; i < n; i++)
    a[i] = 0;
    for (j = 0; j < n; j++) {
        sum = i + j;
        size++;
}
```

- First for loop: n times
- Nested for loops: n^2 times
- Total: O($n + n^2$) = O(n^2)

Switch Case Statement

```
char key;
int X[5], Y[5][5], i, j;
...
switch(key) {
  case 'a' :
      for (i = 0; i < sizeof(X)/sizeof(X[0]); i++)
          sum = sum + X[i];                  => O(n)
      break;
  case 'b' :
      for (i = 0; i < sizeof(Y)/sizeof(Y[0]); i++)
          for (j = 0; j < sizeof(Y[0])/sizeof(Y[0][0]); j++)
              sum = sum + Y[i][j];           => O(n^2)
      break;
} // End of switch block
```

- So using switch statement rule: O(n^2)

for & if else

```
char key;
int A[5][5], B[5][5], C[5][5];
...
if(key == '+') {
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      C[i][j] = A[i][j] + B[i][j];
} // End of if block              => O(n^2)
else if(key == 'x')
  C = matrixMult(A, B);           => O(n^3)
else
  printf("Error! Enter '+' or 'x'! :");   => O(1)
```

- Overall complexity is: O(n^3).

Exponential Algorithm are Expensive

Exercise 6

Let us first prove $n^k = O(b^n)$ whenever $0 < k \leq c$,

Solution

$$\lim \frac{n^k}{b^n} = \lim \frac{kn^{k-1}}{\ln b . b^n} \text{ (set } b^n = e^{n \ln b})$$

- The numerator's exponent decremented after each application of L Hospital's rule.
- So, b^n dominates n^k for any finite k.

Big Oh for Recursive Algorithms

```
procedure T(n: size of the problem) {
    if (n < 1)
        exit()
    Do work of amount n^k

    T(n/b)  // Repeat for a times
    T(n/b)
    ...
    T(n/b)
}
```

- The original problem is recursively divided into a subproblems of n/b.
- In each recursive call O(n^k) work is done.

Master Theorem

- The expression for time complexity is

$$T(n) = aT(n/b) + O(n^k), \text{ where } a > 0, b > 1 \text{ and } k \geq 0$$

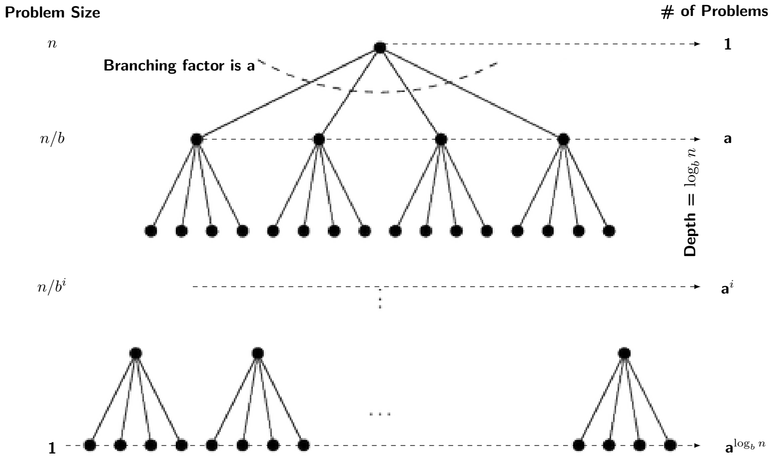
- The time complexity for recursive algorithms is given by:

$$T(n) = \begin{cases} O(n^k) \text{ if } a < b^k \\ O(n^k \log n) \text{ if } a = b^k \\ O(n^{\log_b a}) \text{ if } a > b^k \end{cases}$$

Recursion Tree

- Before solving, let us take a look at recursion tree.
- n is assumed to be a power of b, if not pad n to be larger.
- It requires more than b to be added to n.
- At level 0, when we start the problem size is n.
- At level 1, we have a problems of size n/b each.
- In general, at level i, we have a^i problems of size n/b^i each.

Recursion Tree



Solution of Master's Theorem

- First let us unfold the recurrence relation:

$$
\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + n^k \\
&= a\left(aT\left(\frac{n}{b^2}\right) + \frac{n^k}{b^k}\right) + n^k \\
&\vdots \\
&= n^k + \frac{a}{b^k}n^k + \frac{a^2}{(b^k)^2}n^k + \cdots + \frac{a^L}{(b^k)^L}n^k \\
&= n^k\left(1 + \frac{a}{b^k} + \left(\frac{a}{b^k}\right)^2 + \left(\frac{a}{b^k}\right)^3 + \cdots + \left(\frac{a}{b^k}\right)^L\right)
\end{aligned}
$$

- Here, $L = \log_b n$.

Solution of Master's Theorem: Case I

- The expression within brackets is a GP, of the form

$$1 + r + r^2 + r^3 + \cdots + r^L, \text{ where } r = \frac{a}{b^k} \text{ and } L = \log_b n$$

- In this case $a < b^k$, $r = \frac{a}{b^k} < 1$
- Therefore, the first term dominates the running time.
- In other words, the level 0 of the recursion dominates the runtime.
- Hence, the solution in this case will be O(n^k).

Solution of Master's Theorem: Case II

- In this case $a = b^k$, or $r = 1$ in the expression for the running time.
- In this case, equal work (=n^k) is done at every level of the recursion.
- Since depth of recursion is $1 + \log n$, the running time in this case is O($n^k \log n$).

Solution of Master's Theorem: Case III

- Here, $a > b^k$, which implies $\frac{a}{b^k} > 1$.
- This means the last term in the sum dominates the runtime.
- So, the runtime should be O($n^k \left(\frac{a}{b^k}\right)^L$) = O($a^L$), as $(b^k)^L = (b^L)^k = n^k$
- Now replace L by $\log_b n$ to get O($a^{\log_b n}$)
- $a^L = (b^{\log_b a})^{\log_b n} = (b^{\log_b n})^{\log_b a}$ = $n^{\log_b a}$.