Analyzing Algorithms Intro

Proof for Functions

Formal Definition of Big O: $f(n)$ is $O(g(n))$ iff there exist constants $c>0$ and $n_0>1$ such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$.
Meaning eventually (at a large input) $f(n)$ wont grow faster than any multiple of $g(n)$, basic the only thing that matters is the order of growth

Prove $3n+5$ is $O(n)$:
Find C and n_0 such that $f(n) \le c \cdot g(n)$ for $n \ge n_0$, in this case $g(n) = n$ (the order we're proving is $O(n)$)

$3n+5 \le C \cdot n$ for $n \ge n_0$

$5 \le C \cdot n - 3n$

$\frac{5}{n} \le \frac{n(c-3)}{n}$

$\frac{5}{n} \le c-3$ → Analyze: since $\frac{5}{n}$ is a rational it will be ≤ 1 for $n=5$, for $c=4$, this inequality will be $\frac{5}{n} \le 1$ for all $n \ge 5$
$\therefore c=4, n_0=5$ $\therefore f(n)$ is $O(n)$

Prove $5n^2+3n$ is $O(n^2)$

$5n^2+3n \le C \cdot n^2$ for all $n \ge n_0$

$3n \le c \cdot n^2 - 5n^2$

$\frac{3n}{n^2} \le \frac{n^2(c-5)}{n^2}$

$\frac{3n}{n^2} \le c-5$

$\frac{3}{n} \le c-5$ → Analyze: Holds for $n \ge 3$ and $c=6$. $n_0 = 3, c=6$

Proof for Algorithms

Using Symbolic Constants

```
LinearSearch(L, n, K)
    i = 0                              // C1

    while (i<n) and (L[i] ≠ K)         // C2
        i = i+1

    if i=n then return -1              // C3
    else return i
```

Step 1:

Assign Symbolic constants to lines that execute in constant time

$i=0 \to C_1$ (happens once)
while loop $\to C_2$ (constant time operations)
if $i=n \to C_3$ (happens once)

Step 2: count loop iterations for worst case

Worst case: K is not in list

loop conditions: $i<n \wedge L[i] \neq K$

Since K isn't found. Loop will iterate n times

Step 3: Calculate Runtime

$C_1 = 1$ time
$C_2 = n$ times
$C_3 = 1$ time

$\therefore f(n) = C_1 + C_2 \cdot n + C_3$
$f(n) = (C_1 + C_3) + C_2 \cdot n$
Dominant term: n

This algorithm has $O(n)$ runtime

Example 2: Nested Loop

```
NestedExample(n)
    for i=1 to n:      C1
        for j=1 to n:  C2
            print(i,j)     C3
```

The outer loop will run n times
The inner loop will run n times for each time the outer loop runs
Meaning it will run $n \times n = n^2$ times
print executes n^2 times

$\therefore f(n) = n \cdot C_1 + n^2 \cdot C_2 + n^2 \cdot C_3$ → Order of growth is n^2, algorithm runtime is $O(n^2)$

Example 3:

```
TriangleLoop(n)
  for i=1 to n : c1
    for j=1 to i : c2
      print(i,j) c3
```

Outer loop runs n times

inner loop runs $1+2+3+\ldots+n$ times (1 more time each outer loop)

print is called $\frac{n(n+1)}{2}$ times

$$f(n) = C_1 \cdot n + C_2 \cdot \frac{n(n+1)}{2} + C_3 \cdot \frac{n(n+1)}{2}$$ → This is $O(n^2)$

Logarithmic Complexity:

```
BinarySearch(L, n, x)
  left = 0
  right = n-1

  While left <= right:        c1
    mid = left + right / 2
    if L[mid] == x return mid
    if L[mid] < x : left = mid + 1
    if L[mid] > x : right = mid - 1
  return -1
```

Worst case, x is not in the list

After each iteration, the number of elements will be halved, $n/2$

After 2: $n/4$

After k: $n/2^k$ (2^k = doubling for each iteration)

Lets say after k iterations there is 1 element left $\frac{n}{2^k} = 1$.

$(2^k)\frac{n}{2^k} = 1\,(2^k)$

$= n = 2^k$

$= \log_2(n) = k$

The value of k (number of loop iterations) is $\log_2(n)$

$f(n) = C_1 \log_2(n)$ → $O(\log n)$

Recurrence

Step 1 Define Cost function:

$f(n)$ = number of primitive operations an algorithm does in the worst case for input size n

Using Binary Search example (recursive version)

Worst Case: x not in array

Base case for recursion: when search area is empty: if first > last return 1

Write this as $f(0) = C_1$ (when $n=0$, a constant C_1 amount of work occurs)

Step 2 Cost when $n>0$

When search space isnt empty the algorith:

1. compute mid
2. perform array comparisons
3. call recursion on half the array

This is all constant work. Call it all C_2

This represents a recursive call where the input is halved. (Using $n-1$ here for precision, compared to the previous proof)

$f(n) = C_2 + f((n-1)/2)$ for $n>0$

Step 3 Unroll

Substitute it into itself

$f(n) = C_2 + f\left(\frac{n-1}{2}\right)$ replace n with $\frac{n-1}{2}$

$f\left(\frac{n-1}{2}\right) = C_2 + C_2 + f\left(\frac{(n-1)/2 - 1}{2}\right)$ → Each substitution represents a recursive call using C_2 amount of constant work, so add C_2 each time

Combine constants and simplify

$f\left(\frac{n-1}{2}\right) = 2C_2 + f\left(\frac{n-1-2}{2^2}\right)$

Substitute Again

$$\left(\frac{\left(\frac{(n-1-2)}{2^2}\right) - 1}{2}\right) \Rightarrow \frac{\frac{(n-1-2-2^2)}{2^2}}{2} \Rightarrow \frac{(n-1-2-2^2)}{2^3}$$

$f\left(\frac{n-1-2}{2^2}\right) = 2C_2 + C_2 + f\left(\frac{\left(\frac{(n-1-2)}{2^2}\right) - 1}{2}\right)$

$f\left(\frac{n-1-2}{2^2}\right) = 3C_3 + f\left(\frac{n-1-2-2^2}{2^3}\right)$

↳ Identify Pattern: $f\left(\frac{n - 2^0 - 2^1 - 2^2}{2^3}\right)$ After 3 Subs

→ Simplify geometric Series: $1+2+2^2+2^3+\ldots 2^{k-1}$

Common ratio: $r = \frac{2}{1} = 2$
first term: $a = 1$

use $a \cdot \frac{r^k - 1}{r-1}$

Derive formula for sum: $S = 1+2+2^2+2^3+\ldots+2^{k-1}$

multiply both sides by r: $2S = 2+2^2+2^3+\ldots+2^k$

Subtract $2S - S$: $(2+2^2+2^3+\ldots+2^k) - (1+2+2^2+2^3+\ldots+2^{k-1})$

Every term cancels except: $\boxed{2^k - 1}$

$\therefore f(k) = kC_3 + f\left(\frac{n-(2^k-1)}{2^k}\right)$

Recursion stops in worst case when search space ≤ 1

$(2^k)\,\frac{n-(2^k-1)}{2^k} \leq 1\ (2^k)$

$= n-(2^k-1) \leq 2^k$

$= n - 2^k + 1 \leq 2^k$

$= n+1 \leq 2^k + 2^k$

$= n+1 \leq 2^{k+1}$

$= \log_2(n+1) \leq k+1$

$= \log_2(n+1) - 1 \leq k$

$k = \log(n+1) - 1$ → disregard constants

$k = \log(n)$

$\therefore k$ is the point where recursion ends, thus the worst case run time of this algorithm is $O(\log n)$