Analysis Of Algorithms

The field of computer science that studies efficiency of algorithms is known as **analysis of algorithms**.

Example 1:

Bubblesort requires n^2 statements to sort n elements for a given time interval, t.

If computing speed is increased by 1,000,000, then $1{,}000{,}000n^2$ statements can be executed in the same time interval t.

Since $1{,}000{,}000n^2 = (1000n)^2$

This shows that *only* $1000n$ elements can be sorted which is a $\sqrt{}$ ***increase*** (i.e.,we lost 3 of 6 orders of magnitude)

Ex**ample 2:**

Swap down algorithm in *heapsort* requires *lgn* statements for n elements for a given time interval *t*.

If computing speed increased by 16, then 16lg*n* statements can be executed in the same time interval *t*.

But, $16\lg n = \lg n^{16}$ [since $\lg m^n = n \lg m$]

i.e. n^{16} elements can be processed, which is an ***exponential increase***

NOTE: The amount of speedup that can be realized for an algorithm by using a *faster* machine is proportional to the *inverse* function of the speedup.

Algorithms for which the problem size, which can be solved in a given time interval ***increase linearly*** with computing speed are called linear complexity algorithms.

This information has been summarized in the table below:

Table 1: Effect of algorithm efficiency on ability to benefit from increases in speed

Algorithm Complexity	Linear	Quadratic	Logarithmic	Exponential
Problem size	n	n	n	n
Approx instruction count, time *t*	n	n^2	lg*n*	2^n
Speed increased by 1,000,000, instruction count, time *t*	$1000000n = 10^6 n$	$1000000\ n^2 = 10^6\ n^2$	$1000000 \lg n = \lg n^{1,000,000}$	$10000002^n \approx 2^{20*}2^n = 2^{n+20}$
New Problem size	1,000,000n	1,000n	$n^{1,000,000}$	n + 20
Benefit from increases in speed	**Linear**	$\sqrt{}$	**Exponential**	**Logarithmic**

Complexity of Algorithm Depends On Algorithm Implementation:

Example 1: Bubblesort

Code:

```
for i:= 1 to n-1                          -----line 1
        for j:= 1 to n-1                  -----line 2
                if X[j] > X[j+1]          -----line 3
                        temp := X[j]      -----line 4
                        X[j] := X[j+1]    -----line 5
                        X[j+1] := temp    -----line 6
```

Statement #	# of executions
1	n
2	(n-1)n
3	(n-1)(n-1)
4, 5, 6	$\frac{(n-1)(n-1)}{2}$

$$T(n) = n + (n^2 - n) + (n^2 - 2n + 1) + 3\frac{(n-1)(n-1)}{2}$$

$$= 2n^2 - 2n + 1 + \frac{3n^2 - 6n + 3}{2}$$

$$= \frac{4n^2 - 4n + 2}{2} + \frac{3n^2 - 6n + 3}{2}$$

$$= \frac{7n^2 - 13n + 5}{2} = \boxed{\frac{7}{2}n^2 - \frac{13n}{2} + \frac{5}{2}}$$

Modify line 2 of code to: `j:=1 to n – i` *{ eliminate unneeded comparisons}*

Statement #	# of executions
1	n
2	$\frac{n(n+1)}{2}-1=\sum_{i=2}^{n} i$
3, 4, 5, 6	$\frac{n(n-1)}{2}=\sum_{i=1}^{n-1} i$

$$T(n) = n+\frac{n^2+n}{2}-1+2(n^2-n)$$

$$= (2n^2+n-2n-1)+\frac{n^2+n}{2}$$

$$= \frac{4n^2-2n-2}{2}+\frac{n^2+n}{2}$$

$$= \frac{5n^2-n-2}{2} = \boxed{\frac{5}{2}n^2 - \frac{1}{2}n - 1}$$

Thus, we see as algorithm implementation will vary, T(n) will vary.

Some Definitions:

Characteristic Operation: An algorithm's ***core s***tatement performed repeatedly; This statement is the key to the algorithm's effectiveness and correctness, and is used for finding algorithmic time complexity T(n).

Time Complexity: Number of *characteristic operations* performed

A characteristic operation and its corresponding time complexity function are called **realistic** if the execution time with respect to some implementation is bounded by a linear function of T(n).

Example: Choosing characteristic operation of Bubblesort

Statement # chosen as characteristic operation	Time complexity
1	n
2	(n-1)n
3	(n-1)(n-1)
4, 5, 6	0 to **(**n-1)(n-1) depending on condition

We've seen that for Bubblesort, $T(n) \in O(n^2)$

Statement #1 can be rejected because its complexity is not $O(n^2)$.

From the table above, time complexity is bound by T(n). The choice for the characteristic operation is either statement # 2 or statement # 3.

But statement # 2 (*for j:= 1 to n-1*) is a looping construct that does not have any inherent connection to the meaning of the algorithm.

Statement # 3 (*if X*[j] > *X*[j+1]), on the other hand is the ***core*** of the algorithm and hence a good choice for characteristic operation of the algorithm

Another variation of bubblesort code:

```
pass := 1                                   -----line 1
swap := true                                -----line 2
while swap and (pass < n-1)                 -----line 3
        swap := false                       -----line 4
        for j := 1 to (n – pass)            -----line 5
                if X[j] > X[j + 1]          -----line 6
                        temp := X[j]        -----line 7
                        X[j] := X[j+1]      -----line 8
                        X[j+1] := temp      -----line 9
                        swap := true        -----line 10
```

Here, number of passes varies from 1 to (n – 1) depending on contents of array.

Best, Worst and Average Time Complexities

Algorithm P accepts 'k' different instances of size 'n'
Let

$T_i(n)$ is time complexity of P given i^{th} instance $(1 \le i \le k)$
P_i is probability that this instance occurs

Then,

Best-case Complexity,

$$B(n) = \underset{1 \le i \le k}{Min} T_i(n)$$

Worst-case Complexity,

$$W(n) = \underset{1 \le i \le k}{Max} T_i(n)$$

Average-case Complexity,

$$A(n) = \sum_{i=1}^{k} p_i T_i(n)$$

Example: Linear Search

Code:

```
LinearSearch (var R: RA type; a, b, x: integer): boolean
    var i integer; found: boolean;
    i := a;
    found := false
    while (i ≤ b ) and not found
        found := R[i] = = x        <-- characteristic operation
        i++
    LinearSearch := found
```

Algorithm has infinite instances, but all can be categorized into (n+1) classes:
x = R[1], x = R[2], x = R[3], … x = R[n], x ∉ R

i	**Instance**	**p_i**	**$T_i(n)$**
1	x = R[1]	p/n	1
2	x = R[2]	p/n	2
i	x = R[i]	p/n	i
n	x = R[n]	p/n	n
n+1	x ∉ R[1… n]	1 – p	n

Here,

B(n) = 1

W(n) = n (occurs in two cases)

$$A(n) = \sum_{i=1}^{n+1} p_i T_i(n)$$

$$= \sum_{i=1}^{n} p_i T_i(n) + p_{n+1} T_{n+1}(n)$$

$$= \sum_{i=1}^{n} \frac{p}{n} i + (1-p)n$$

$$= p \sum_{i=1}^{n} \frac{i}{n} + (1-p)n$$

$$= \frac{p}{n} \frac{(n)(n+1)}{2} + (1-p)n$$

$$= \boxed{\frac{(p)(n+1)}{2} + (1-p)n}$$

Variation: Linear Search with Ordered Array (early stop possible)

Here,

i	**Instance**	**p_i**	**$T_i(n)$**	
1	x = A[1]	p/n	1	
2	x = A[2]	p/n	2	
i	x = A[i]	p/n	i	
n	x = A[n]	p/n	n	**success**
n+1	x < A[1]	(1 – p)/n	1	
n+2	x < A[2]	(1 – p)/n	2	
j	x < A[j]	(1 – p)/n	j	**failure**
2n-1	x < A[2n-1]	(1 – p)/n	n-1	
2n	x < A[2n]	(1 – p)/n	n	

For linear search with ordered array,

$$A(n) = \sum_{i=1}^{2n} p_i T_i(n)$$

$$= \sum_{i=1}^{n} p_i T_i(n) + \sum_{i=n+1}^{2n} p_i T_i(n) = \sum_{i=1}^{n} \frac{p}{n} i + \sum_{i=n+1}^{2n} \frac{(1-p)}{n}(i-n)$$

$$= \frac{p}{n}[\frac{(n)(n+1)}{2}] + (\frac{1-p}{n}) \sum_{i=i+1}^{2n} (i-n)$$

$$= \frac{p}{n}[\frac{(n)(n+1)}{2}] + \frac{(1-p)}{n} \sum_{i=1}^{n} i$$

$$= \frac{(p)(n+1)}{2} + \frac{(1-p)(n+1)}{2} = \boxed{\frac{(n+1)}{2}}$$

Q. When is early stop better than no early stop in unordered array?

The above occurs when

	$A_{UNORDERED}(n)$	>	$A_{EARLY\ STOP}(n)$
or,	$\frac{(p)(n+1)}{2}+(1-p)n$	>	$\frac{(n+1)}{2}$
or,	$\frac{(p)(n+1)+2n(1-p)}{2}$	>	$\frac{(n+1)}{2}$
or,	$\frac{pn+p+2n-2pn}{2}$	>	$\frac{(n+1)}{2}$
or,	$p+2n-pn$	>	$n+1$
or,	$p-pn$	>	$1-n$
or,	$p(1-n)$	>	$1-n$

or, **p < 1** **(we assume n >> 1 → sign changes by division)**

This shows that with our assumption that whenever Pr(element in array) < 1.0 EARLY STOP is better.

Analysis of Recursive Algorithms

Let T(n) ≡ time complexity of (recursive) algorithm
Time complexity can be defined recursively.

Example 1: Factorial Function

```
Code:  int factorial (int n) {
           if (n = = 0) return 1;
           else return n * factorial (n – 1);
       }
```

Recurrence Equation:

$$T(n) = T(n-1) + 1; \quad n > 0 \quad \{\text{all } n > 0\}$$
$$T(0) = 1 \quad ; \quad n = 0 \quad \{\text{base case}\}$$

Thus,

$$T(n-1) = T(n-2) + 1$$
$$T(n-2) = T(n-3) + 1$$
$$T(n-3) = T(n-4) + 1$$
$$\vdots \qquad \vdots$$

There are a variety of techniques for solving the recurrence equation (to solve for T(n) without T(n-1) on right side of the equation**)** to get a **closed form**. We'll start with the simplest technique: **Repeated Substitution**

Solving the recurrence equation of the factorial form to get the closed form,

$$T(n) = T(n-1) + 1$$
$$= [T(n-2) + 1] + 1$$
$$= [[T(n-3) + 1] + 1] + 1$$
$$= \ldots\ldots$$
$$= T(n-i) + i \qquad \{i^{th} \text{ level of substitution}\}$$

if 'i = n',

$$T(n) = T(0) + n$$

or, $\boxed{T(n) = n + 1}$

Example 2: Towers of Hanoi

Code:
```
procedure Hanoi (n: integer; from, to, aux: char)
    if n > 0
        Hanoi (n – 1, from, aux, to)
        write ('from', from, 'to', to)
        Hanoi (n – 1, aux, to, from)
```

Here,

$T(0) = 0$

$T(n) = T(n-1) + 1 + T(n-1); \quad n > 0$

$= 2T(n-1) + 1$

Solving the recurrence equation to get the closed form using repeated substitution,

$T(n) = 2T(n-1) + 1$

$T(n-1) = 2T(n-2) + 1$

$T(n-2) = 2T(n-3) + 1$

$T(n-3) = 2T(n-4) + 1$

$T(n) = 2T(n-1) + 1$

$= 2[2T(n-2) + 1] + 1$

$= 2[2[2T(n-3) + 1] + 1] + 1$

$= 2[2^2T(n-3) + 2^1 + 2^0] + 2^0$

$= 2^3T(n-3) + 2^2 + 2^1 + 2^0$

Repeating 'i' times,

$T(n) = 2^iT(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 2^0$

If 'i = n',

$T(n) = 2^nT(0) + 2^{n-1} + 2^{n-2} + \ldots + 2^0$

$= 2^{n-1} + 2^{n-2} + \ldots + 2^0$

$= \sum_{i=0}^{n-1} 2^i$

This is a geometric series whose sum is given by: $S_n = \frac{a - ar^n}{1 - r}$

Here, a = 1, r = 2, n = n. Therefore, $S_n = \frac{1 - 2^n}{1 - 2} = 2^n - 1$.

i.e. $\boxed{T(n) = 2^n - 1}$

Example 3: Binary Search

Code:

```
function BinSearch (var R: Ratype; a,b: indextype; x: keytype): boolean
    var mid: indextype
        if a > b
            BinSearch:= false
        else
            mid:= (a + b) div 2
            if x = R[mid]                    characteristic operation
                BinSearch:= true
            else
                if x < R[mid]
                BinSearch:=BinSearch(R,a, mid–1,x)
                else
                BinSearch:=BinSearch(R,mid+1,b,x)
```

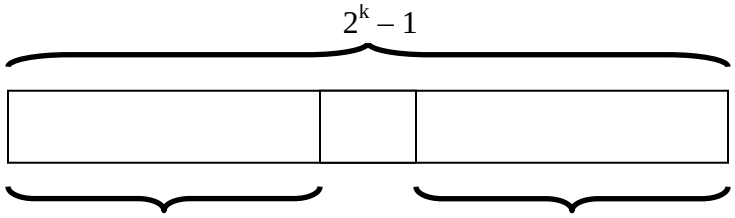
$2^{k-1} - 1$ $2^{k-1} - 1$

$$T(0) = 0$$

$$T(n) = \begin{cases} 1 & \text{if } x = R[mid] \\ 1 + T(\ (n+1)/2\ - 1) & \text{if } x < R[mid] \\ 1 + T(n - \ (n+1)/2\ - 1) & \text{if } x > R[mid] \end{cases}$$

Complexity of BinSearch(R, 1,n, x)

We can eliminate the floor function by limiting 'n' to $2^k - 1$; $k \in Z^+$



Now,

$T(0) = 0$

$$T(n = 2^k\text{-}1) = \begin{cases} 1 & \text{if } x = R[mid] \\ 1 + T(2^{k-1} - 1) & \text{if } x \neq R[mid] \end{cases}$$

Best case occurs if x = R[mid] is true the very first time.

Thus,

$$B(n) = 1$$

Worst case occurs if x = R[mid] is always false.

Now,

$W(0) = 0$

$$W(2^k - 1) = \quad 1 + W(2^{k-1} - 1)$$

Solving the recurrence equation to get the closed form using repeated substitution,

$W(2^k - 1) = 1 + W(2^{k-1} - 1)$
$= 1 + [1 + W(2^{k-2} - 1)]$
$= 1 + [1 + [1 + W(2^{k-3} - 1)]]$

Repeating 'i' times,

$W(2^k - 1) = i + W(2^{k-i} - 1)$

if 'i = k',

$W(2^k - 1) = k + 0 = k$

We want W(n), so

$2^k - 1 = n$
or, $\lg 2^k = \lg (n + 1)$
or, $k = \lg (n + 1)$ {k in terms of n}

Thus,

$W(n) = \lg (n + 1)$

Average Case Complexity

First we need to set up a model.

As a first step we'll determine the average case complexity given that the element is present in the array. Here, we take into account the probability of not finding an element at a particular level (in worst case we assumed had taken this part to be one i.e. the element being not found at a particular level was taken as a certainty) and add to it the work done at the next level recursively.

Thus,

$A_{ElementPresent}(1) = 1$

$$A_{ElementPresent}(2^k - 1) = (1 - \frac{1}{2^k - 1}) + A(2^{k-1} - 1)$$

$$= (1 - \frac{1}{2^k - 1}) + [(1 - \frac{1}{2^{k-1} - 1}) + A(2^{k-2} - 1)]$$

$$= 1 - \frac{1}{2^k - 1} + 1 - \frac{1}{2^{k-1} - 1} + 1 - \frac{1}{2^{k-2} - 1} + A(2^{k-3} - 1)$$

Repeating 'i' times,

$$A_{ElementPresent}(2^k - 1) = [i] - \frac{1}{2^k - 1} - \frac{1}{2^{k-1} - 1} - \frac{1}{2^{k-2} - 1} + \ldots + \frac{1}{2^{k-(i-1)} - 1} + A(2^{k-i} - 1)$$

Let 'i = k-1',

$$A_{ElementPresent}(2^k - 1) = [k - 1] + 1 - \sum_{i=0}^{k-2} \frac{1}{2^{k-i} - 1}$$

$$= k - \sum_{i=0}^{k-2} \frac{1}{2^{k-i} - 1}$$

However, $\sum_{i=0}^{k-2} \frac{1}{2^{k-i}-1}$ is a strictly increasing series, therefore its upper bound is given by

$$\sum_{i=0}^{k-2} \frac{1}{2^{k-i}-1} \le \int_0^{k-1} \frac{1}{2^{k-x}-1} dx \approx \int_0^{k-1} \frac{1}{2^{k-x}} dx$$

$$= \int_0^{k-1} 2^{x-k}\, dx = \frac{2^{x-k}}{\ln 2}\Big|_0^{k-1}$$

$$= \frac{1}{2\ln 2} - \frac{2^{-k}}{\ln 2}$$

Substituting, k = lg (n + 1) we have

$$\frac{1}{2\ln 2} - \frac{2^{-k}}{\ln 2} = \frac{1}{2\ln 2} - \frac{2^{-\lg(n+1)}}{\ln 2}$$

$$= \frac{1}{2\ln 2} - \frac{1}{\ln 2 \bullet 2^{\lg(n+1)}} = \frac{1}{2\ln 2} - \frac{1}{\ln 2(n+1)}$$

Now combining both the terms we have,

$A_{ElementPresent}$ (n) = $\lg(n+1) - \frac{1}{2\ln 2} + \frac{1}{\ln 2(n+1)}$ note: $\frac{1}{2\ln 2} \cong 0.7213$

We can consider all cases where the element may or may not be in the array ….

Given that: $p \equiv$ Pr (element exist in the array)

$$A(n) = p(\text{average cost when element is present}) + (1 - p)\ (\text{worst cost})$$
$$= p[\lg(n+1) - \frac{1}{2\ln 2} + \frac{1}{\ln 2(n+1)}] + (1 - p)\ [\lg(n+1)]$$

For example, if n = 1023,

$$A(n) = p[\lg(1024) - 0.7213 + \frac{1}{0.6932(1024)} + (1 - p)[\lg(1024)$$
$$= p(9.3) + (1 - p)(10)$$

Solving Homogenous Linear Recurrence Relations Using Method of Characteristic Roots (Another Method To Solve Recurrence Equations)

Homogenous Linear Recurrence Relation:

Example 1:

Recurrence Relation: $a_n = 5a_{n-1} - 6a_{n-2}$
$a_1 = 7$

$a_0 = 1$

General form:
$a_n + c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k} = 0$
$a_{k-1} = \ldots$
.
.
$a_0 = \ldots$

Let $a_n = r^n \forall n$
Substituting this in original recurrence relation, we have

$r^n = 5r^{n-1} - 6r^{n-2}$

or, $r^n - 5r^{n-1} + 6r^{n-2} = 0$

Dividing by r^{n-2} gives us,

$r^2 - 5r + 6 = 0$

or, $(r - 2)(r - 3) = 0$

=> $r_1 = 2 \quad r_2 = 3$

=> $a_n = 2^n$ or, $a_n = 3^n$; both work

Therefore, **general solution** is: $\boxed{a_n = \alpha 2^n + \beta 3^n}$

Now, consider the initial conditions:

$a_1 = 7$ or, $\alpha 2^1 + \beta 3^1 = 7$ or, $2\alpha + 3\beta = 7$ ----- (1)

$a_0 = 1$ or, $\alpha 2^0 + \beta 3^0 = 1$ or, $\alpha + \beta = 1$ or, $\alpha = 1 - \beta$ ----- (2)

Substituting value of α from (2) in (1), we get

$2(1-\beta) + 3\beta = 7$

$2 - 2\beta + 3\beta = 7$

$2 + \beta = 7$

$\beta = 5$

Therefore,

$\alpha = 1 - \beta = 1 - 5$

$or, \alpha = -4$

Thus, **specific solution** is: $\boxed{a_n = 5 \bullet 3^n - 4 \bullet 2^n}$

Cross-checking some values,

$a_0 = 5 \bullet 3^0 - 4 \bullet 2^0 = 1$

$a_1 = 5 \bullet 3^1 - 4 \bullet 2^1 = 7$

$a_2 = 5 \bullet 3^2 - 4 \bullet 2^2 = 29$

.

.

$a_n = 5 \bullet 3^n - 4 \bullet 2^n = \ldots$

These are same as

$a_n = 5a_{n-1} - 6a_{n-2}$

$a_0 = 1; and, a_1 = 7;$

$a_2 = 5 \bullet a_1 - 6 \bullet a_2$

$or, a_2 = 5 \bullet 7 - 6 \bullet 1 = 29$

Example 2: Fibonacci Sequence

Recurrence Relation: $a_n = a_{n-1} - a_{n-2}$

$a_1 = 1$

$a_0 = 0$

Let $a_n = r^n \forall n$

Substituting this in original recurrence relation, we have

$$r^n = r^{n-1} + r^{n-2}$$

or, $$r^n - r^{n-1} - r^{n-2} = 0$$

Dividing by r^{n-2} gives us,

$$r^2 - r - 1 = 0$$

=> $$r_1 = \frac{1+\sqrt{5}}{2} \quad r_2 = \frac{1-\sqrt{5}}{2}$$

=> $a_n = (\frac{1+\sqrt{5}}{2})^n$ or, $a_n = (\frac{1-\sqrt{5}}{2})^n$; both work

Therefore, **general solution** is:

$$a_n = \alpha(\frac{1+\sqrt{5}}{2})^n + \beta(\frac{1-\sqrt{5}}{2})^n$$

Now, consider the initial conditions:

$a_1 = 1$ or, $\alpha(\frac{1+\sqrt{5}}{2})^1 + \beta(\frac{1-\sqrt{5}}{2})^1 = 1$ ----- (1)

$a_0 = 0$ or, $\alpha + \beta = 0$ or, $\alpha = -\beta$ ----- (2)

Substituting value of α from (2) in (1), we get

$$-\beta(\frac{1+\sqrt{5}}{2}) + \beta(\frac{1-\sqrt{5}}{2}) = 1$$
$$-\beta(1+\sqrt{5}) + \beta(1-\sqrt{5}) = 2$$
$$-\beta + (-\beta\sqrt{5}) + \beta + (-\beta\sqrt{5}) = 2$$
$$-2\beta\sqrt{5}) = 2$$
$$-\beta\sqrt{5}) = 1$$
$$\beta = -\frac{1}{\sqrt{5}}$$
$$\therefore \alpha = \frac{1}{\sqrt{5}}$$

Thus, **specific solution** is: $\boxed{a_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n}$

Crosschecking values,

$$a_0 = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^0 - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^0 = \frac{1}{\sqrt{5}} - \frac{1}{\sqrt{5}} = 0$$
$$a_1 = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^1 - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^1 = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}-1+\sqrt{5}}{2}) = \frac{2\sqrt{5}}{2\sqrt{5}} = 1$$

Example 3:

Recurrence Relation: $a_n = a_{n-1} + 6a_{n-2}$
$a_1 = 8$
$a_0 = 1$

Let $a_n = r^n \forall n$
Substituting this in original recurrence relation, we have

$r^n = r^{n-1} + 6r^{n-2}$

or, $r^n - r^{n-1} - 6r^{n-2} = 0$

Dividing by r^{n-2} gives us,

$r^2 - r - 6 = 0$

=> $r_1 = -2 \quad r_2 = 3$

=> $a_n = (-2)^n$ or, $a_n = 3^n$; both work

Therefore, **general solution** is: $\boxed{a_n = \alpha(-2)^n + \beta 3^n}$

Now, consider the initial conditions:

$a_1 = 8$ or, $-2\alpha + 3\beta = 8$ ----- (1)

$a_0 = 1$ or, $\alpha + \beta = 1$ or, $\alpha = 1 - \beta$ ----- (2)

Substituting value of α from (2) in (1), we get

$-2(1-\beta) + 3\beta = 8$

$-2 + 5\beta = 8$

$5\beta = 10$

$\beta = 2$

$\therefore \alpha = 1 - 2 = -1$

Thus, **specific solution** is: $a_n = -1(-2)^n = 2 \bullet 3^n$

Crosschecking values,

$a_0 = -1(-2)^0 + 2(3)^0 = -1 + 2 = 1$

$a_1 = -1(-2)^1 + 2(3)^1 = 2 + 6 = 8$

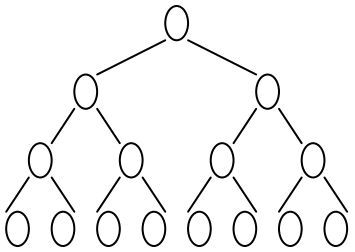
$a_2 = -1(-2)^2 + 2(3)^2 = -4 + 18 = 14$

$a_3 = -1(-2)^3 + 2(3)^3 = 8 + 54 = 62$

Iterative Algorithms

Just as recursive algorithms lead naturally to recurrence equations, so iterative algorithms lead naturally to formulas involving summations. The following examples use an iterative approach to solve the given problem.

Example 1: Average Depth of BST Node



First we need to set up a model
Let $n = 2^k - 1 \approx 2^k$
Given,

n = 16, k = 4

n is the number of nodes

k is the height of the tree

Approach 1

One way to look at it is that half the total number of nodes have depth 3, one-fourth of total nodes have depth 2 and so on. Thus, total depth of all the nodes can be put in the form:

$$\frac{1}{2} \bullet 3 + \frac{1}{4} \bullet 2 + \frac{1}{8} \bullet 1 + \frac{1}{16} \bullet 0$$

or,

$$\sum_{i=1}^{k} \frac{1}{2^i}(k-i) = \sum_{i=1}^{k} \frac{k}{2^i} - \sum_{i=1}^{k} \frac{i}{2^i}$$

$$= k\sum_{i=1}^{k} 2^{-i} - \sum_{i=1}^{k} i2^{-i}$$

This gives us a model but we can find the formula. However, this approach uses approximate values. The second approach we'll see uses exact values and hence gives a better model.

Approach 2

Another way to look at it is that 1 node has depth 0, 2 nodes have depth 1 and so on. Here, the total depth of all the nodes is given by the form:

$$2^0 \bullet 0 + 2^1 \bullet 1 + 2^2 \bullet 2 + 2^3 \bullet 3$$

or,

$$\sum_{i=0}^{k-1} i2^i = \sum_{i=1}^{k-1} i2^i$$

But we know,

$$\sum_{i=1}^{k} i2^i = (k-1)2^{k+1} + 2$$ **(proof given in handout: 'Mathematical Tools', Page 4)**

Substituting this value in the equation above, we have

$$\sum_{i=1}^{k-1} i2^i = (k-2)2^k + 2$$

$$= (\log_2(n+1) - 2)(n+1) + 2 \quad [\text{since } n = 2^k - 1]$$

$$\text{Average Depth of full BST Node} = \frac{(\log_2(n+1) - 2)(n+1) + 2}{n}$$

The following table shows some actual values of average depths of BST nodes for given values of k:

k	n	average depth
4	15	2.27
5	31	3.16
6	63	4.10
7	127	5.06
8	255	6.03
9	511	7.02
10	1023	8.01
11	2047	9.01
12	4095	10.00
13	8191	11.00
14	16383	12.00
15	32767	13.00
16	65535	14.00

Example 2: Binary Search

Using iterative approach, the following model can be set up for the average number of comparisons for binary search work.

Assume the total number of elements in an array, $n = 2^k - 1$

Then,

$$\Pr(\text{comparisons} = 1) = \frac{1}{2^k - 1} = \frac{2^0}{2^k - 1}$$

$$\Pr(\text{comparisons} = 2) = \frac{2}{2^k - 1} = \frac{2^1}{2^k - 1}$$

.

.

$$\Pr(\text{comparisons} = k) = \frac{2^{k-1}}{2^k - 1}$$

Therefore, assuming element is in array

$$\text{Average number of comparisons} = \sum_{comp} \#ofComparisons * \Pr(\#comparisons)$$

$$= \sum_{i=1}^{k} i \bullet \frac{2^{i-1}}{2^k - 1}$$

$$= \frac{\sum_{i=1}^{k} i \bullet 2^{i-1}}{2^k - 1}$$

Solving the numerator, we have

$$\sum_{i=1}^{k} i \bullet 2^{i-1} = \sum_{i=1}^{k} i[2^i - 2^{i-1}]$$

$$= \sum_{i=1}^{k} i2^i - \sum_{i=1}^{k} i2^{i-1} = \sum_{i=1}^{k} i2^i - \sum_{i=0}^{k-1} (i+1)2^i$$

$$= \sum_{i=1}^{k} i2^i - \sum_{i=0}^{k-1} i2^i - \sum_{i=0}^{k-1} 2^i$$

$$= k2^k - 2^k + 1$$

$$= (k-1)2^k + 1$$

This is a geo. prog. where,
a = 1
r = 2
n = k

Therefore, $\mathbf{S_n} = \frac{1 - 2^k}{1 - 2}$

Putting the entire expression together, we have

$$\text{Average number of comparisons}_{\textbf{(assuming element is in array)}} = \frac{(k-1)2^k + 1}{2^k - 1}$$

$$= \frac{(\lg(n+1)-1)(n+1)+1}{(n+1)-1}$$
$$= \frac{(n+1)\lg(n+1)-n-1+1}{n}$$
$$= \frac{(n+1)\lg(n+1)-n}{n}$$

or,

Average number of comparisons**(assuming element is in array)**	$\frac{(n+1)\lg(n+1)}{n}-1$

As $n \to \infty$, the complexity approaches $\mathbf{lg(n+1)-1}$ [This is the large N solution]

For example, if n = 15 ($2^4 - 1$) i.e. k = 4

Numeric solution is given by: $\sum_{i=1}^{4} i \bullet 2^{i-1} = \frac{1+4+12+32}{15} = \frac{49}{15} = 3.2\overline{66}$

Application of analytical solution gives us: $\frac{(4-1)2^4+1}{2^4-1}-1 = \frac{49}{15} = 3.2\overline{66}$ ***or,***

$$\frac{(15+1)\lg(15+1)2^4}{15}-1 = \frac{(16)(4)}{15}-1 = \frac{49}{15} = 3.2\overline{66}$$