SE2205

Algorithms and Data Structures for Object-Oriented Design

Instructor: Pirathayini Srikantha

Western University

January 16, 2019

Readings/References

- Goodrich (4)

Table of Contents

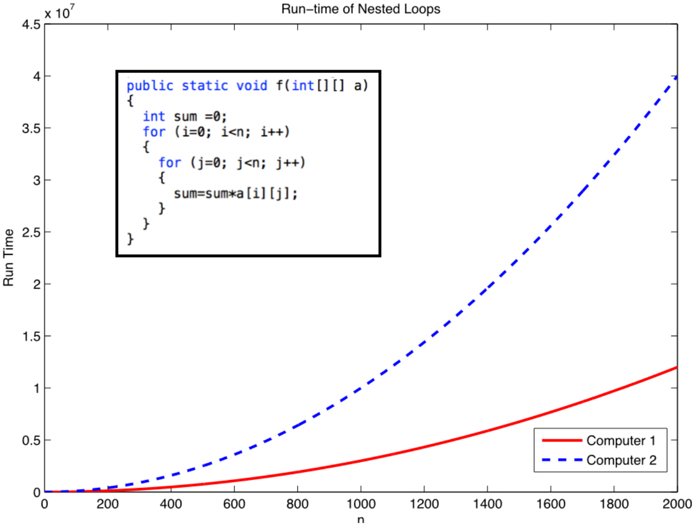
Table of Contents

Motivation

- How can we **quantitatively** characterize the **efficiency** of a program?
  - Can we use clocks?
- Time required for completing the execution of an algorithm depends on:
  - CPU
  - Memory availability/usage
  - Disk usage
  - Network usage
- Need a way to quantify the efficiency of an algorithm that is **independent** of these factors!

Motivation

Assume that we run the following nested for-loop statements on two different computers



Complexity Classes

- From the previous example, it is clear that there is a **common pattern of resource consumption**
    - The run-time of both algorithms are quadratic functions of the data size (i.e. $f(n) = an^2 + bn + c$)
- All programs/algorithms fall into different **complexity classes**
- The relation between run-time and problem size of algorithms in a particular complexity class share the **same basic shape**
    - In the previous example, the curve falls in the *quadratic* complexity class
- Differences between curves in the same complexity class are introduced only by **constant coefficients** (i.e. `a`, `b`, `c`)

O-Notation

- **O-Notation** is used to denote the complexity class of a algorithm: $O(f(n))$
- What complexity class does $f(n) = an^2 + bn + c$ fall under? $O(an^2 + bn + c)$
- What is the fastest growing term in $f(n)$?
  - For algorithms with a large problem size n, the run-time is **dominated** by the fastest growing term
  - Suppose $n = 1000$, $a = 0.5$, $b = 3$, $c = 10$
  - $bn + c$ terms account for 0.5% of $f(n)$ while an^2 accounts for 99.5%
  - $O(an^2)$
- Need a **general curve** to determine the complexity class and therefore the proportionality constant a can be removed
- The complexity class of $f(n)$ is $O(n^2)$ which is the quadratic class
- Despite the differences in resources in computers, the run-time of the program generally grows quadratically with the problem size

Intuitively Why is the Code Snippet Quadratic?

```
public static void f(int[][] a)
{
  int sum =0;
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      sum=sum*a[i][j];
    }
  }
}
```

Intuitively Why is the Code Snippet Quadratic?

```
public static void f(int[][] a)
{
   int sum =0;
   for (i=0; i<n; i++)
   {
     for (j=0; j<n; j++)
     {
       sum=sum*a[i][j];
     }
   }
}
```

How many times is the multiplication operation executed?

Intuitively Why is the Code Snippet Quadratic?

```
public static void f(int[][] a)
{
  int sum =0;
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      sum=sum*a[i][j];
    }
  }
}
```

n multiplication operations

Intuitively Why is the Code Snippet Quadratic?

```
public static void f(int[][] a)
{
  int sum =0;
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      sum=sum*a[i][j];
    }
  }
}
```

How many times is the outer loop executed?

Intuitively Why is the Code Snippet Quadratic?

```
public static void f(int[][] a)
{
  int sum =0;
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      sum=sum*a[i][j];
    }
  }
}
```

n times!

The outer loop executes n times. This means that the inner loop is executed n times as well.

The inner loop executes n times. Overall, there are n*n operations!

Hence, the run-time is quadratic.

Popular Complexity Classes

- O(1): constant time algorithm (for $n = 100$, $f(100) = 1$)
  - Algorithm is not affected by problem size
- O(log(n)): logarithmic time algorithm (for $n = 100$, $f(100) = 2$)
- O(n): linear time algorithms (for $n = 100$, $f(100) = 100$)
- O(nlogn): (for $n = 100$, $f(100) = 200$)
- $O(n^2)$: quadratic time algorithms (for $n = 100$, $f(100) = 10000$)
- $O(n^3)$: cubic time algorithms (for $n = 100$, $f(100) = 1000000$)
  - matrix multiplication
- $O(2^n)$: exponential algorithms (for $n = 100$, $f(100) = 1.27 * e^{30}$)

Table of Contents

Formal Definition of O-Notation

Definition

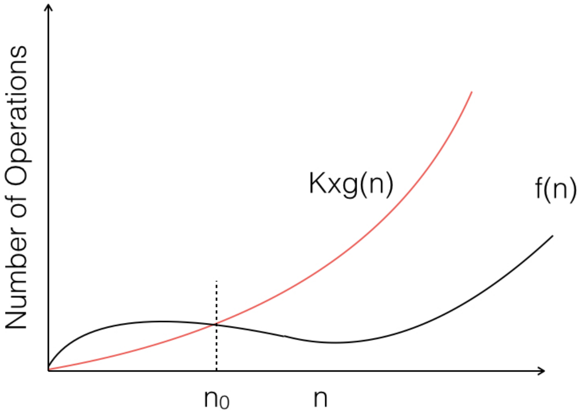
O-Notation: $f(n)$ is O(g(n)) if there exist two positive constants K and n_0 such that $|f(n)| \leq K|g(n)| \; \forall \; n \geq n_0$

Formal definition in simpler terms:

- Consider a sufficiently large problem $n \geq n_0$ and assume that $g(n) \geq 0$ and $f(n) \geq 0 \; \forall \; n$
- If an algorithm runs in $O(g(n))$ then it runs to completion in no more than a constant K multiplied by $(g(n))$ time steps/operations
- For all values of $n \geq n_0$, if the curve $K * g(n)$ is an upper bound of $f(n)$ then $g(n)$ is the complexity class that $f(n)$ falls under

Graphical Definition of O-Notation

- Consider a sufficiently large problem $n \geq n_0$
- If an algorithm runs in $O(g(n))$ then it runs to completion in no more than a constant (K) multiplied by abs$(g(n))$ time steps.



O-Notation

- O-notation relates the cost of **efficiency** (i.e. number of operations, time, space) of an algorithm/program with the **size of the problem**
- O-notation is an **asymptotic** notation as it describes the behaviour of an algorithm for large problem sizes (i.e. $n \to \infty$)
- $O(g(n))$ represents all functions that are asymptotically bounded above by $K * |g(n)|$
- This measure is **independent** of highly varying features such as clocks, CPUs, memory, etc.
- Although typically the O-notation is used to represent **worst** case scenarios, it is also possible to obtain the O-notation for the **best** and **average** case scenarios

Shortcuts to Finding O(g(n))

Suppose the cost of executing an algorithm is $f(n)$

- Separate terms in $f(n)$ into dominant and lesser terms:

$$f(n) = ((constant * \text{dominant term}) + \text{lesser terms})$$

- How to distinguish dominancy of terms in a relation?
- Use the scale of strength:

$$O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^3) < O(2^n) < O(10^n)$$

- Eliminate the lesser terms and the constant coefficients:

$$O(f(n)) = O((\cancel{\text{constant}} * \text{dominant term}) + \cancel{\text{lesser terms}}))$$

  - Constant terms include bases of logs (i.e. $log_{10}(n) = log_2(n)/log_2(10)$)
- These manipulations result in $O(f(n)) = dominant\ term$

Why are these Shortcuts Justified?

- Let $f(n) = ((constant * \text{dominant term}) + \text{lesser terms})$
- The following holds:

$$\text{lesser term} < dominant\ \ term$$

- Assume that P is the number of lesser terms in $f(n)$

$$\text{lesser terms} < P * dominant\ \ term$$

- Let $P = K - constant$ and add $constant*$dominant terms to both sides

$$\begin{aligned} constant * \text{dominant term} < \quad & (K - constant) * dominant\ \ term \\ + \text{lesser terms} \quad & +constant * \text{dominant term} \end{aligned}$$

- Result is $f(n) < K * dominant\ \ term$ which is $f(n) < K * g(n)$
- This is precisely the definition of $f(n) \in O(g(n))$

Implied Assumptions for Defining O(g(n))

- Is there a unique O(g(n))? No! The definition of $O(g(n))$ does not impose any uniqueness
- Any K*g(n) that is an upper bound to f(n) when $n \to \infty$ is a valid complexity class representing f(n)
- For instance, if $f(n) \in O(n^2)$ then the following is true as well: $f(n) \in O(n^3)$

A1 A good practice is to set the bound to be as tight as possible: i.e. $f(n) \in O(n^2)$

A2 While it is good practice to set a tight bound, ensure that $g(n)$ is also as simple as possible

- For instance suppose that $O(n^2 + nlog(n))$ is a tight bound on $f(n)$
- However, according to [A2] it is good practise to simplify this and set $f(n) \in O(n^2)$

Example

- Suppose the cost of executing an algorithm is quantified as

$$f(n) = 5 + 10 + 15 + ... + 5n = 5(1 + 2 + 3 + ... + n)$$

- Looks familiar? This is an **arithmetic progression**!
- S is a sequence of this form:

$$\begin{aligned} S =& a + (a + d) + (a + 2d) + \ldots + (l - 2d) + (l - d) + l \\ S =& n(a + l)/2 \end{aligned}$$

- Applying this formula to $f(n)$:

$$\begin{aligned} f(n) &= 5n(1 + n)/2 = 5n/2 + 5n^2/2 \\ O(f(n)) &= O(n^2) \end{aligned}$$

- **Formal Proof:** Need to show that $|f(n)| \leq K|g(n)| \ \forall n \geq n_0$

$$\begin{aligned} &(5n + 5n^2)/2 \leq (5n^2 + 5n^2)/2 = 5n^2 \\ &n \leq n^2 \\ &n \geq 1 \quad \square \text{ where } K = 5 \ n_0 = 1 \end{aligned}$$

Table of Contents

Other Asymptotic Notations

There exist other asymptotic notations:

- $\Omega(l(n))$: asymptotic lower bound
- $\Theta(h(n))$: asymptotic tight bound

Definition

O-Notation: $f(n) \in O(g(n))$ if there exist two positive constants K and n_0 such that $|f(n)| \leq K|g(n)|\ \forall\ n \geq n_0$

Definition

Ω-Notation: $f(n) \in \Omega(l(n))$ if there exist two positive constants L and n_0 such that $|f(n)| \geq L|l(n)|\ \forall\ n \geq n_0$

Definition

Θ-Notation: If $f(n) \in O(h(n))$ and $f(n) \in \Omega(h(n))$ then $f(n) \in \Theta(h(n))$ (i.e. $\exists\ L$ and K $s.t.$ $L|h(n)| \leq |f(n)| \leq K|h(n)|$)

Illustrations of Asymptotic Notations

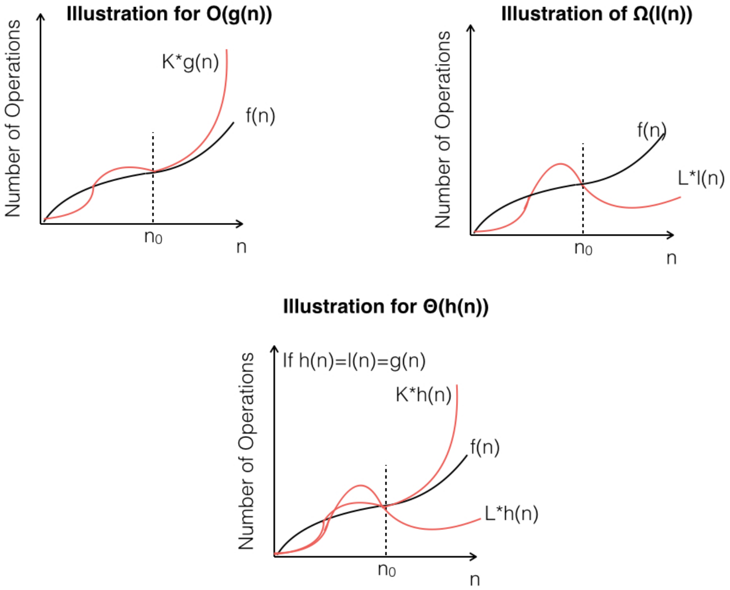
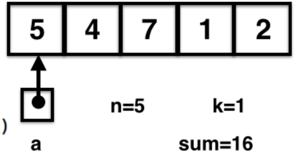


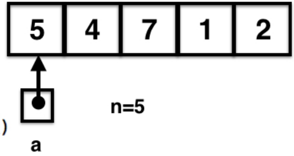
Table of Contents

Example #1




```
public int sumSubset(int[] a, int n, int k)
{
  int i;
  int sum=0;
  for (i=0; i<n; i++) {
    sum=sum+a[i];
    if (k==a[i]) {
      return sum;
    }
  }
  return -1;
}
```

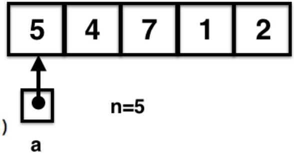
Example #1




```
public int sumSubset(int[] a, int n, int k)
{
  int i;
  int sum=0;
  for (i=0; i<n; i++) {
    sum=sum+a[i];
    if (k==a[i]) {
      return sum;
    }
  }
  return -1;
}
```

Fixed number of operations
Cost is constant say ‘c’

Example #1

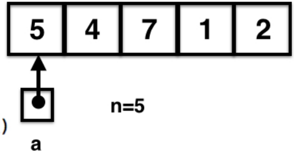



```
public int sumSubset(int[] a, int n, int k)
{
  int i;
  int sum=0;
  for (i=0; i<n; i++) {
    sum=sum+a[i];
    if (k==a[i]) {
      return sum;
    }
  }
  return -1;
}
```

Fixed number of operations
Cost is constant say ‘c’

Fixed number of operations
Cost is constant say ‘b’

Example #1




```java
public int sumSubset(int[] a, int n, int k)
{
  int i;
  int sum=0;
  for (i=0; i<n; i++) {
    sum=sum+a[i];
    if (k==a[i]) {
      return sum;
    }
  }
  return -1;
}
```

Fixed number of operations
Cost is constant say ‘c’

Fixed number of operations
Cost is constant say ‘b’

This loop may terminate before i=n-1

Cost is constant say ‘a’

Example #1: What is the complexity in the best case?

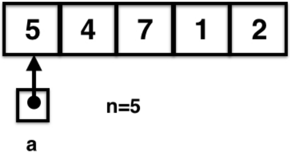
```
public int sumSubset(int[] a, int n, int k)
{
  int i;
  int sum=0;
  for (i=0; i<n; i++) {
    sum=sum+a[i];
    if (k==a[i]) {
      return sum;
    }
  }
  return -1;
}
```

Cost: c

Cost: b

Cost: a

Cost: a



- The best case occurs when k is at the beginning of the array
- In this case, the run-time is $f(0) = c + b + a$
- Since $f(0)$ is a constant, $O(f(0)) = O(1)$

Example #1: What is the complexity in the worst case?

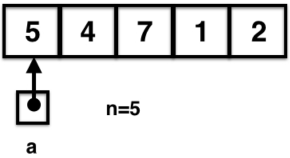
```java
public int sumSubset(int[] a, int n, int k)
{
   int i;
   int sum=0;
   for (i=0; i<n; i++) {
     sum=sum+a[i];
     if (k==a[i]) {
       return sum;
     }
   }
   return -1;
}
```

Cost: c

Cost: b

Cost: a

Cost: a



- When k is at the end of the array
- For-loop will go through each element in the array
- Cost in the worst case:

$$f(n) = b * n + c + a$$
$$O(f(n)) = O(n)$$

- Worst case run-time is linear

Supp. Example #1: Complexity in the average case?

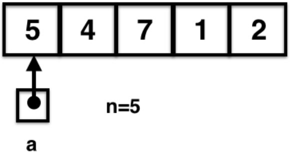
```java
public int sumSubset(int[] a, int n, int k)
{
  int i;
  int sum=0;
  for (i=0; i<n; i++) {
    sum=sum+a[i];
    if (k==a[i]) {
      return sum;
    }
  }
  return -1;
}
```

Cost: c

Cost: b

Cost: a

Cost: a



- The probability of a key occurring at index i can be assumed to be equal (i.e. $p(i) = 1/n$)
- The cost of k occurring at index i: $f(i) = b(i+1) + c$
- Expected cost of a random variable: $E(i) = \sum_{i=0}^{n-1} p(i) f(i)$

Supp. Example #1: Complexity in the average case?

$$E(i) = \sum_{i=0}^{n-1} p(i)f(i) = \sum_{i=0}^{n-1} \frac{1}{n}(b(i+1)+c) <= \textbf{Let } \boldsymbol{j = i+1}$$

$$= \sum_{j=1}^{n} \frac{1}{n}(bj+c) = \frac{b}{n}\sum_{j=1}^{n} j + \frac{c}{n}\sum_{j=1}^{n} 1$$

- In the first term $\sum_{j=1}^{n} j$ is an arithmetic progression: $S = (1+2+\ldots+n) = \frac{n(n+1)}{2}$
- Second term $\sum_{j=1}^{n} 1$ is n

$$E(i) = \frac{b}{2}(n+1) + c$$

- $E(i)$ is the average cost!
- $O(E(i))$ is $O(n)$ and therefore the average run-time is linear

Example #2

```java
public int[][] fillLowerTriangle(int n)
{
  int i,j;
  int[][] a=new int[n][n];
  for (i=0; i<n; i++) {
    for (j=0; j<i; j++) {
      a[i][j]=i;
      System.out.print(a[i][j]);
    }
    System.out.println("");
  }
  return a;
}
```

Program Output:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

Example #2

```java
public int[][] fillLowerTriangle(int n)
{
  int i,j;
  int[][] a=new int[n][n];
  for (i=0; i<n; i++) {
    for (j=0; j<i; j++) {
      a[i][j]=i;
      System.out.print(a[i][j]);
    }
    System.out.println("");
  }
  return a;
}
```

Cost: a

Cost: c

Cost: d

Cost: e

Example #2

```
public int[][] fillLowerTriangle(int n)
{
                 Cost a
  for (i=0; i<n; i++) {
    for (j=0; j<i; j++) {
                 Cost c
    }
                 Cost d
  }
                 Cost e
}
```

- Before the first for-loop cost is a
- Cost of every iteration in the innermost nested for-loop is c
- Cost of every iteration in the outermost nested for loop is $ci + d$
- Cost of the entire nested for-loop is $\sum_{i=0}^{n-1}(ci + d)$
- Cost of the last statement is e

Example #2

```
public int[][] fillLowerTriangle(int n)
{
                Cost a
  for (i=0; i<n; i++) {
    for (j=0; j<i; j++) {
                Cost c
    }
                Cost d
  }
                Cost e
}
```

- Overall complexity is $f(n)$

$$
\begin{aligned}
f(n) &= a + \sum_{i=0}^{n-1} (ci + d) + e \\
&= a + \frac{n(n-1)c}{2} + dn + e \\
&= a + e + dn + \frac{cn^2}{2} - \frac{nc}{2}
\end{aligned}
$$

- O(f(n))=O(n^2)

Example #3: Analyzing Recursive Algorithms

```
public int power(int base, int exp)
{
  int res=0;
  if (exp==0) {
    res=1;
  }
  else{
    res=power(base, exp-1);
    res=res*base;
  }
  return res;
}
```

Example #3: Analyzing Recursive Algorithms

```java
public int power(int base, int exp)
{
   int res=0;
   if (exp==0) {
      res=1;
   }
   else{
      res=power(base, exp-1);
      res=res*base;
   }
   return res;
}
```

Cost: a

Cost: b

Cost: f(n-1)

Cost: c

Cost: d

- Note the cost assigned to the recursive function call
- It is the cost of executing the recursive call of a smaller problem i.e. $f(n-1)$ where n is *exp*

Example #3: Analyzing Recursive Algorithms

```
public int power(int base, int exp)
{
              Cost: a

              Cost: b
   }
   else{
            Cost: f(n-1)
              Cost: c
   }
              Cost: d
}
```

- Overall cost when $n > 0$ is

$$f(n) = a + f(n-1) + c + d, \text{ let } a + c + d = e$$
$$f(n) = e + f(n-1)$$

- This is a recurrence relation!
- Recurrence relations require base cases: $f(0) = a + b + d$, let $a + b + d = g$
- Solve the recurrence problem:

$$f(0) = g$$
$$f(n) = e + f(n-1)$$

Example #3: Analyzing Recursive Algorithms

- Need to solve the following complete recurrence relation problem:

$$\begin{aligned} f(0) &= g \\ f(n) &= e + f(n-1) \end{aligned}$$

- Use the method of *unrolling* (i.e. expand the recurrent term)

$$\begin{aligned} f(0) &= g \\ f(n) &= e + f(n-1) \textbf{<= f(n-1)=e+f(n-2)} \\ &= e + e + f(n-2) \\ &= ne + f(0) \\ &= ne + g \end{aligned}$$

- It is clear that O(f(n)) = O(n)

Example #4: Towers of Hanoi

```
public static void moveTowers(int start, int spare, int finish, int n){
  if(n==1)
    System.out.println("Move disk from "+start+" to "+finish);
  else{
    moveTowers(start, finish, spare, n-1);
    System.out.println("Move disk from "+start+" to "+finish);
    moveTowers(spare, start, finish, n-1);
  }
}
```

Example #4: Towers of Hanoi

```java
public static void moveTowers(int start, int spare, int finish, int n){
  if(n==1)
    System.out.println("Move disk from "+start+" to "+finish);
  else{
    moveTowers(start, finish, spare, n-1);
    System.out.println("Move disk from "+start+" to "+finish);
    moveTowers(spare, start, finish, n-1);
  }
}
```

Cost: a

Cost: F(n-1)

Cost: b

Cost: F(n-1)

Example #4: Towers of Hanoi

```
public static void moveTowers(int start, int spare, int finish, int n){
  if(n==1)
     Cost: a
  else{
     Cost: F(n-1)
     Cost: b
     Cost: F(n-1)
  }
}
```

- Cost of the base case is a
- Cost of the first recursive call is $F(n-1)$
- Cost of the statement in between the two recursive calls is b
- Cost of the second recursive call is $F(n-1)$

$$F(1) = a$$
$$F(n) = b + 2F(n-1)$$

Example #4: Towers of Hanoi

- Need to solve the following recurrence relation:

$$F(1) = a$$
$$F(n) = b + 2F(n-1)$$

- Geometric sum is useful

$$S_n = \sum_{i=0}^{n} ar^i = a\frac{1 - r^{n+1}}{1 - r}$$

- Rolling out the equations:

$F(n) = b + 2F(n-1)$

Example #4: Towers of Hanoi

- Need to solve the following recurrence relation:

$$F(1) = a$$
$$F(n) = b + 2F(n-1)$$

- Geometric sum is useful

$$S_n = \sum_{i=0}^{n} ar^i = a\frac{1 - r^{n+1}}{1 - r}$$

- Rolling out the equations:

$$\begin{aligned} F(n) &= b + 2F(n-1) \\ &= b + 2(b + 2F(n-2)) = 2^0 b + 2^1 b + 2^2 F(n-2) \end{aligned}$$

Example #4: Towers of Hanoi

- Need to solve the following recurrence relation:

$$F(1) = a$$
$$F(n) = b + 2F(n-1)$$

- Geometric sum is useful

$$S_n = \sum_{i=0}^{n} ar^i = a\frac{1-r^{n+1}}{1-r}$$

- Rolling out the equations:

$$\begin{aligned} F(n) &= b + 2F(n-1) \\ &= b + 2(b + 2F(n-2)) = 2^0 b + 2^1 b + 2^2 F(n-2) \\ &= 2^0 b + 2^1 b + 2^2(b + 2F(n-3)) = 2^0 b + 2^1 b + 2^2 b + 2^3 F(n-3)) \end{aligned}$$

Example #4: Towers of Hanoi

- Need to solve the following recurrence relation:

$$F(1) = a$$
$$F(n) = b + 2F(n-1)$$

- Geometric sum is useful

$$S_n = \sum_{i=0}^{n} ar^i = a\frac{1 - r^{n+1}}{1 - r}$$

- Rolling out the equations:

$$\begin{aligned}
F(n) &= b + 2F(n-1) \\
&= b + 2(b + 2F(n-2)) = 2^0 b + 2^1 b + 2^2 F(n-2) \\
&= 2^0 b + 2^1 b + 2^2(b + 2F(n-3)) = 2^0 b + 2^1 b + 2^2 b + 2^3 F(n-3)) \\
&= 2^0 b + 2^1 b + \ldots 2^{n-2} b + 2^{n-1} F(1)
\end{aligned}$$

Example #4: Towers of Hanoi

- Need to solve the following recurrence relation:

$$F(1) = a$$
$$F(n) = b + 2F(n-1)$$

- Geometric sum is useful

$$S_n = \sum_{i=0}^{n} ar^i = a\frac{1 - r^{n+1}}{1 - r}$$

- Rolling out the equations:

$$\begin{aligned}
F(n) &= b + 2F(n-1) \\
&= b + 2(b + 2F(n-2)) = 2^0 b + 2^1 b + 2^2 F(n-2) \\
&= 2^0 b + 2^1 b + 2^2(b + 2F(n-3)) = 2^0 b + 2^1 b + 2^2 b + 2^3 F(n-3)) \\
&= 2^0 b + 2^1 b + \ldots 2^{n-2} b + 2^{n-1} F(1) \\
&= 2^0 b + 2^1 b + \ldots 2^{n-2} b + 2^{n-1} a = \sum_{i=0}^{n-2} 2^i b + 2^{n-1} a
\end{aligned}$$

Example #4: Towers of Hanoi

- Need to solve the following recurrence relation:

$$F(1) = a$$
$$F(n) = b + 2F(n-1)$$

- Geometric sum is useful

$$S_n = \sum_{i=0}^{n} ar^i = a\frac{1 - r^{n+1}}{1 - r}$$

- Rolling out the equations:

$$\begin{aligned}
F(n) &= b + 2F(n-1) \\
&= b + 2(b + 2F(n-2)) = 2^0 b + 2^1 b + 2^2 F(n-2) \\
&= 2^0 b + 2^1 b + 2^2(b + 2F(n-3)) = 2^0 b + 2^1 b + 2^2 b + 2^3 F(n-3)) \\
&= 2^0 b + 2^1 b + \ldots 2^{n-2} b + 2^{n-1} F(1) \\
&= 2^0 b + 2^1 b + \ldots 2^{n-2} b + 2^{n-1} a = \sum_{i=0}^{n-2} 2^i b + 2^{n-1} a \\
&= (2^{n-1} - 1)b + 2^{n-1} a = 2^{n-1}(b + a) - b
\end{aligned}$$

Example #4: Towers of Hanoi

- What is the O(F(n))?

$$F(n) = 2^{n-1}(b + a) - b$$

- $O(F(n))=O(2^n)$

Example #5: Recursive Algorithm

```java
public int func(int[] list, int n)
{
  int[] lista,listb;
  if(n>1){
    copy(lista, list, 0, n/2);
    copy(listb, list, n/2+1, n);
    func(lista, n/2);
    func(listb, n/2);
    organize(list, lista, listb, n);
  }
}
```

Cost: a

Cost: dn+e

Cost: 2F(n/2)

Cost: gn+h

- Cost of the base case is $F(1) = a$
- Cost of the non-base case is $F(n) = a + dn + e + 2F(n/2) + gn + h$
- Let $a + h = c$ and $d + g = b$ and therefore $F(n) = bn + c + 2F(n/2)$
- Recurrence relation to be solved is

$$F(1) = a$$
$$F(n) = bn + c + 2F(n/2)$$

Example #5: Recursive Algorithm

- Unrolling the recurrence relation:

$$\begin{aligned} F(n) &= bn + c + 2F(n/2) = bn + c + 2(bn/2 + c + 2F(n/4)) \\ &= 2bn + c + 2c + 4(bn/4 + c + 2F(n/8)) \end{aligned}$$

- $O(F(n)) = n \log_2 n$

Example #5: Recursive Algorithm

- Unrolling the recurrence relation:

$$\begin{aligned} F(n) &= bn + c + 2F(n/2) = bn + c + 2(bn/2 + c + 2F(n/4)) \\ &= 2bn + c + 2c + 4(bn/4 + c + 2F(n/8)) \\ &= 3bn + c + 2c + 4c + 8F(n/8) \end{aligned}$$

- $O(F(n)) = n\log_2 n$

Example #5: Recursive Algorithm

- Unrolling the recurrence relation:

$$\begin{aligned} F(n) &= bn + c + 2F(n/2) = bn + c + 2(bn/2 + c + 2F(n/4)) \\ &= 2bn + c + 2c + 4(bn/4 + c + 2F(n/8)) \\ &= 3bn + c + 2c + 4c + 8F(n/8) \\ &= 3bn + 2^0c + 2^1c + 2^2c + 2^3F(n/2^3) \quad \textbf{<= Let } \boldsymbol{n = 2^K} \end{aligned}$$

- $O(F(n)) = n\log_2 n$

Example #5: Recursive Algorithm

- Unrolling the recurrence relation:

$$\begin{aligned}
F(n) &= bn + c + 2F(n/2) = bn + c + 2(bn/2 + c + 2F(n/4)) \\
&= 2bn + c + 2c + 4(bn/4 + c + 2F(n/8)) \\
&= 3bn + c + 2c + 4c + 8F(n/8) \\
&= 3bn + 2^0c + 2^1c + 2^2c + 2^3F(n/2^3) \quad \color{red}{\mathbf{<= Let}\ \boldsymbol{n = 2^K}} \\
&= Kbn + \sum_{i=0}^{K-1} 2^i c + 2^K F(2^K/2^K)
\end{aligned}$$

- $O(F(n)) = n\log_2 n$

Example #5: Recursive Algorithm

- Unrolling the recurrence relation:

$$
\begin{aligned}
F(n) &= bn + c + 2F(n/2) = bn + c + 2(bn/2 + c + 2F(n/4)) \\
&= 2bn + c + 2c + 4(bn/4 + c + 2F(n/8)) \\
&= 3bn + c + 2c + 4c + 8F(n/8) \\
&= 3bn + 2^0c + 2^1c + 2^2c + 2^3F(n/2^3) \;\; \mathbf{<= Let\ } \boldsymbol{n = 2^K} \\
&= Kbn + \sum_{i=0}^{K-1} 2^i c + 2^K F(2^K/2^K) \\
&= Kbn + c\frac{1-2^K}{1-2} + 2^K a \;\; \mathbf{<=} \boldsymbol{\log_2 n = K}
\end{aligned}
$$

- $O(F(n)) = n\log_2 n$

Example #5: Recursive Algorithm

- Unrolling the recurrence relation:

$$
\begin{aligned}
F(n) &= bn + c + 2F(n/2) = bn + c + 2(bn/2 + c + 2F(n/4)) \\
&= 2bn + c + 2c + 4(bn/4 + c + 2F(n/8)) \\
&= 3bn + c + 2c + 4c + 8F(n/8) \\
&= 3bn + 2^0c + 2^1c + 2^2c + 2^3F(n/2^3) \quad \textcolor{red}{\mathbf{<= \text{Let } n = 2^K}} \\
&= Kbn + \sum_{i=0}^{K-1} 2^i c + 2^K F(2^K/2^K) \\
&= Kbn + c\frac{1-2^K}{1-2} + 2^K a \quad \textcolor{red}{\mathbf{<= log_2 n = K}} \\
&= bn\log_2 n + c(n-1) + an
\end{aligned}
$$

- $O(F(n)) = n\log_2 n$

Example #5: Recursive Algorithm

- Unrolling the recurrence relation:

$$
\begin{aligned}
F(n) &= bn + c + 2F(n/2) = bn + c + 2(bn/2 + c + 2F(n/4)) \\
&= 2bn + c + 2c + 4(bn/4 + c + 2F(n/8)) \\
&= 3bn + c + 2c + 4c + 8F(n/8) \\
&= 3bn + 2^0c + 2^1c + 2^2c + 2^3F(n/2^3) \; \color{red}{\boldsymbol{<= \text{Let } n = 2^K}} \\
&= Kbn + \sum_{i=0}^{K-1} 2^i c + 2^K F(2^K/2^K) \\
&= Kbn + c\frac{1-2^K}{1-2} + 2^K a \; \color{red}{\boldsymbol{<= log_2 n = K}} \\
&= bn\log_2 n + c(n-1) + an \\
&= bn\log_2 n + (a+c)n - c
\end{aligned}
$$

- $O(F(n)) = n\log_2 n$

Table of Contents

General Classes of Recurrence Relations

- There are three general classes of recurrence relations:
- Class 1:

$$
\begin{aligned}
F(1) &= a \\
F(n) &= b + cF(n-1)
\end{aligned}
$$

- Class 2:

$$
\begin{aligned}
F(1) &= a \\
F(n) &= bn + c + dF(n-1)
\end{aligned}
$$

- Class 3:

$$
\begin{aligned}
F(1) &= a \\
F(n) &= bn + c + dF(n/p)
\end{aligned}
$$

- Can express these as non-recurrent relations

Useful Mathematical Tools:

- **Arithmetic sequence**:

$$S = \sum_{i=0}^{n-1}(a + id) = \frac{n}{2}(2a + (n-1)d)$$

- **Geometric sequence**:

$$S = \sum_{i=0}^{n-1} ar^i = \frac{a(1 - r^n)}{1 - r}$$

- **Heaviside cover-up** rules for partial fraction expansion:

$$\frac{ax^2 + bx + c}{(x-d)(x-e)^2} = \frac{A}{x-d} + \frac{B}{x-e} + \frac{C}{(x-e)^2}$$

$$A = \frac{ad^2 + bd + c}{(d-e)^2} \quad C = \frac{ae^2 + be + c}{(e-d)}$$

B : **solve for B** $\quad \dfrac{ac^2 + bc + c}{(c-d)(c-e)^2} = \dfrac{A}{c-d} + \dfrac{B}{c-e} + \dfrac{C}{(c-e)^2}$

where c is a constant

Recurrence Relation: Class 1

$$\textbf{General Form: } \begin{aligned} F(1) &= a \\ F(n) &= b + cF(n-1) \end{aligned}$$

Unrolling recurrence relation:

$$\begin{aligned} F(n) &= b + cF(n-1) = b + c(b + cF(n-2)) \\ &= b + cb + c^2F(n-2) = b + cb + c^2b + c^3F(n-3) \end{aligned}$$

Recurrence Relation: Class 1

$$\textbf{General Form: } \begin{aligned} F(1) &= a \\ F(n) &= b + cF(n-1) \end{aligned}$$

Unrolling recurrence relation:

$$\begin{aligned} F(n) &= b + cF(n-1) = b + c(b + cF(n-2)) \\ &= b + cb + c^2F(n-2) = b + cb + c^2b + c^3F(n-3) \\ &= b + cb + c^2b + c^3b + \ldots + c^{n-2}b + c^{n-1}F(1) \end{aligned}$$

Recurrence Relation: Class 1

$$\textbf{General Form: } \begin{aligned} F(1) &= a \\ F(n) &= b + cF(n-1) \end{aligned}$$

Unrolling recurrence relation:

$$\begin{aligned} F(n) &= b + cF(n-1) = b + c(b + cF(n-2)) \\ &= b + cb + c^2F(n-2) = b + cb + c^2b + c^3F(n-3) \\ &= b + cb + c^2b + c^3b + \ldots + c^{n-2}b + c^{n-1}F(1) \\ &= b\sum_{i=0}^{n-2} c^i + c^{n-1}a = b\frac{1-c^{n-1}}{1-c} + c^{n-1}a \end{aligned}$$

Recurrence Relation: Class 1

General Form:
$$F(1) = a$$
$$F(n) = b + cF(n-1)$$

Unrolling recurrence relation:

$$\begin{aligned}
F(n) &= b + cF(n-1) = b + c(b + cF(n-2)) \\
&= b + cb + c^2F(n-2) = b + cb + c^2b + c^3F(n-3) \\
&= b + cb + c^2b + c^3b + \ldots + c^{n-2}b + c^{n-1}F(1) \\
&= b\sum_{i=0}^{n-2} c^i + c^{n-1}a = b\frac{1-c^{n-1}}{1-c} + c^{n-1}a \\
&= \frac{b}{1-c} - \frac{bc^n}{c(1-c)} + \frac{c^n a}{c} \quad \textbf{<=coverup rule } \frac{\boldsymbol{b}}{\boldsymbol{c(1-c)}}
\end{aligned}$$

Recurrence Relation: Class 1

General Form: $F(1) = a$

$$F(n) = b + cF(n-1)$$

Unrolling recurrence relation:

$$\begin{aligned} F(n) &= b + cF(n-1) = b + c(b + cF(n-2)) \\ &= b + cb + c^2F(n-2) = b + cb + c^2b + c^3F(n-3) \\ &= b + cb + c^2b + c^3b + \ldots + c^{n-2}b + c^{n-1}F(1) \\ &= b\sum_{i=0}^{n-2} c^i + c^{n-1}a = b\frac{1-c^{n-1}}{1-c} + c^{n-1}a \\ &= \frac{b}{1-c} - \frac{bc^n}{c(1-c)} + \frac{c^na}{c} \quad \textbf{<=coverup rule } \frac{b}{c(1-c)} \end{aligned}$$

$$F(n) = c^n\left(\frac{a}{c} - \left(\frac{b}{c} + \frac{b}{1-c}\right)\right) - \frac{b}{c-1} \quad \mathbf{<=} \; \frac{b}{c(1-c)} = \frac{b}{c} + \frac{b}{1-c}$$

Recurrence Relation: Class 2

General Form:
$$\begin{aligned} F(1) &= a \\ F(n) &= bn + c + dF(n-1) \end{aligned}$$

Unrolling recurrence relation:

$$\begin{aligned} F(n) &= bn + c + dF(n-1) \\ &= bn + c + d(b(n-1) + c + dF(n-2)) \end{aligned}$$

Recurrence Relation: Class 2

General Form:

$$
\begin{aligned}
F(1) &= a \\
F(n) &= bn + c + dF(n-1)
\end{aligned}
$$

Unrolling recurrence relation:

$$
\begin{aligned}
F(n) &= bn + c + dF(n-1) \\
&= bn + c + d(b(n-1) + c + dF(n-2)) \\
&= bn + c + bd(n-1) + cd + d^2F(n-2)
\end{aligned}
$$

Recurrence Relation: Class 2

$$\textbf{General Form: } \begin{aligned} F(1) &= a \\ F(n) &= bn + c + dF(n-1) \end{aligned}$$

Unrolling recurrence relation:

$$\begin{aligned} F(n) &= bn + c + dF(n-1) \\ &= bn + c + d(b(n-1) + c + dF(n-2)) \\ &= bn + c + bd(n-1) + cd + d^2F(n-2) \\ &= bn + c + bd(n-1) + cd + d^2(b(n-2) + c + dF(n-3)) \end{aligned}$$

Recurrence Relation: Class 2

General Form:
$$F(1) = a$$
$$F(n) = bn + c + dF(n-1)$$

Unrolling recurrence relation:

$$\begin{aligned}
F(n) &= bn + c + dF(n-1) \\
&= bn + c + d(b(n-1) + c + dF(n-2)) \\
&= bn + c + bd(n-1) + cd + d^2 F(n-2) \\
&= bn + c + bd(n-1) + cd + d^2(b(n-2) + c + dF(n-3)) \\
&= bd^0 n + bd^1(n-1) + bd^2(n-2) + c + cd + cd^2 + d^3 F(n-3)
\end{aligned}$$

Recurrence Relation: Class 2

General Form: $F(1) = a$

$$F(n) = bn + c + dF(n-1)$$

Unrolling recurrence relation:

$$\begin{aligned}
F(n) &= bn + c + dF(n-1) \\
&= bn + c + d(b(n-1) + c + dF(n-2)) \\
&= bn + c + bd(n-1) + cd + d^2F(n-2) \\
&= bn + c + bd(n-1) + cd + d^2(b(n-2) + c + dF(n-3)) \\
&= bd^0n + bd^1(n-1) + bd^2(n-2) + c + cd + cd^2 + d^3F(n-3) \\
&= bd^0n + bd^1(n-1) + bd^2(n-3) + \ldots + bd^{n-2}2 \ \textbf{<= S1} \\
&+ c + cd + cd^2 + \ldots + cd^{n-2} \ \textbf{<= S2} \\
&+ d^{n-1}F(1) \ \textbf{<= S3}
\end{aligned}$$

Recurrence Relation: Class 2

$$\textbf{S1:}\quad \begin{aligned} S &= bd^0 n + bd^1(n-1) + bd^2(n-2) + \ldots + bd^{n-2}2 \\ -\,dS &= -\,bd^1 n - bd^2(n-1) - \ldots - bd^{n-2}3 - bd^{n-1}2 \end{aligned}$$

Recurrence Relation: Class 2

$$\begin{aligned}
\textbf{S1:}\ \ S =& bd^0 n + bd^1(n-1) + bd^2(n-2) + \ldots + bd^{n-2}2 \\
- dS =& - bd^1 n - bd^2(n-1) - \ldots - bd^{n-2}3 - bd^{n-1}2 \\
\hline
S(1-d) =& b(n - d - d^2 - \ldots - d^{n-2} - d^{n-1} - d^{n-1})
\end{aligned}$$

Recurrence Relation: Class 2

$$
\begin{aligned}
\textbf{S1:}\quad S &= bd^0 n + bd^1(n-1) + bd^2(n-2) + \ldots + bd^{n-2}2 \\
-dS &= -bd^1 n - bd^2(n-1) - \ldots - bd^{n-2}3 - bd^{n-1}2 \\
\hline
S(1-d) &= b(n - d - d^2 - \ldots - d^{n-2} - d^{n-1} - d^{n-1}) \\
S &= \frac{b(n - d(d^0 + d^1 + \ldots + d^{n-2}) - d^{n-1})}{1-d}
\end{aligned}
$$

Recurrence Relation: Class 2

$$
\begin{aligned}
\textbf{S1:}\quad S =& bd^0 n + bd^1(n-1) + bd^2(n-2) + \ldots + bd^{n-2}2 \\
-dS =& -bd^1 n - bd^2(n-1) - \ldots - bd^{n-2}3 - bd^{n-1}2 \\
\hline
S(1-d) =& b(n - d - d^2 - \ldots - d^{n-2} - d^{n-1} - d^{n-1}) \\
S =& \frac{b(n - d(d^0 + d^1 + \ldots + d^{n-2}) - d^{n-1})}{1-d} \\
S =& \frac{b(n - d(\frac{1-d^{n-1}}{1-d}) - d^{n-1})}{1-d} = b\frac{n}{1-d} - b\frac{d-d^n}{(1-d)^2} - b\frac{d^{n-1}}{1-d}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{S2:}\quad & c(d^0 + d^1 + d^2 + \ldots + d^{n-2}) \\
& c(\sum_{i=0}^{n-2} d^i) = c\frac{1-d^{n-1}}{1-d} = c\frac{d-d^n}{d(1-d)}
\end{aligned}
$$

$$
\textbf{S3:}\quad d^{n-1}F(1) = d^{n-1}a = \frac{d^n a}{d}
$$

Recurrence Relation: Class 2

S1+S2+S3:

$$F(n) = \frac{bn}{1-d} - b\frac{d-d^n}{(1-d)^2} - b\frac{d^{n-1}}{1-d} + c\frac{d-d^n}{d(1-d)} + \frac{d^n a}{d}$$

Recurrence Relation: Class 2

S1+S2+S3:

$$
\begin{aligned}
F(n) =& \frac{bn}{1-d} - b\frac{d-d^n}{(1-d)^2} - b\frac{d^{n-1}}{1-d} + c\frac{d-d^n}{d(1-d)} + \frac{d^n a}{d} \\
=& \frac{bn}{1-d} + \frac{-bd^2 + cd(1-d)}{d(1-d)^2} \\
& + \left(\frac{b}{(1-d)^2} - \frac{c}{d(1-d)} + \frac{a}{d} - \frac{b}{d(1-d)}\right)d^n
\end{aligned}
$$

Recurrence Relation: Class 2

S1+S2+S3:

$$
\begin{aligned}
F(n) =& \frac{bn}{1-d} - b\frac{d-d^n}{(1-d)^2} - b\frac{d^{n-1}}{1-d} + c\frac{d-d^n}{d(1-d)} + \frac{d^n a}{d} \\
=& \frac{bn}{1-d} + \frac{-bd^2 + cd(1-d)}{d(1-d)^2} \\
& + \left(\frac{b}{(1-d)^2} - \frac{c}{d(1-d)} + \frac{a}{d} - \frac{b}{d(1-d)}\right)d^n \\
=& \frac{bn}{1-d} + \frac{c - d(b+c)}{(1-d)^2} \\
& + \left(\frac{b}{(1-d)^2} - \frac{b+c}{d(1-d)} + \frac{a}{d}\right)d^n
\end{aligned}
$$

Recurrence Relation: Class 2

S1+S2+S3:

$$
\begin{aligned}
F(n) =& \frac{bn}{1-d} - b\frac{d-d^n}{(1-d)^2} - b\frac{d^{n-1}}{1-d} + c\frac{d-d^n}{d(1-d)} + \frac{d^n a}{d} \\
=& \frac{bn}{1-d} + \frac{-bd^2 + cd(1-d)}{d(1-d)^2} \\
& + \left(\frac{b}{(1-d)^2} - \frac{c}{d(1-d)} + \frac{a}{d} - \frac{b}{d(1-d)}\right)d^n \\
=& \frac{bn}{1-d} + \frac{c-d(b+c)}{(1-d)^2} \\
& + \left(\frac{b}{(1-d)^2} - \frac{b+c}{d(1-d)} + \frac{a}{d}\right)d^n
\end{aligned}
$$

Cover-up rule can be used to expand $\frac{b+c}{d(1-d)} = \frac{b+c}{d} + \frac{b+c}{1-d}$

$$
F(n) = \frac{bn}{1-d} + \frac{c-d(b+c)}{(1-d)^2} + \left(\frac{b}{(1-d)^2} + \frac{a-b-c}{d} + \frac{b+c}{d-1}\right)d^n
$$

Recurrence Relation: Class 3

General Form:
$$F(1) = a$$
$$F(n) = bn + c + dF(n/p)$$

Unrolling recurrence relation:

$$\begin{aligned} F(n) &= bn + c + dF(n/p) = bn + c + d(bn/p + c + dF(n/p^2)) \\ &= bn + bnd/p + c + cd + d^2(bn/p^2 + c + dF(n/p^3)) \end{aligned}$$

Recurrence Relation: Class 3

General Form: $F(1) = a$

$$F(n) = bn + c + dF(n/p)$$

Unrolling recurrence relation:

$$\begin{aligned}
F(n) &= bn + c + dF(n/p) = bn + c + d(bn/p + c + dF(n/p^2)) \\
&= bn + bnd/p + c + cd + d^2(bn/p^2 + c + dF(n/p^3)) \\
&= bn + bnd/p + bn(d/p)^2 + c + cd + cd^2 + d^3F(n/p^3)
\end{aligned}$$

Recurrence Relation: Class 3

General Form: $F(1) = a$

$$F(n) = bn + c + dF(n/p)$$

Unrolling recurrence relation:

$$F(n) = bn + c + dF(n/p) = bn + c + d(bn/p + c + dF(n/p^2))$$
$$= bn + bnd/p + c + cd + d^2(bn/p^2 + c + dF(n/p^3))$$
$$= bn + bnd/p + bn(d/p)^2 + c + cd + cd^2 + d^3F(n/p^3)$$
$$= bn((\frac{d}{p})^0 + (\frac{d}{p})^1 + (\frac{d}{p})^2 + \ldots + (\frac{d}{p})^{K-1}) \;\; \textbf{<= Let } \boldsymbol{n = p^K}$$
$$+ c(d^0 + d^1 + d^2 + \ldots + d^{K-1}) + d^K F(\frac{n}{p^K})$$

Recurrence Relation: Class 3

General Form:
$$F(1) = a$$
$$F(n) = bn + c + dF(n/p)$$

Unrolling recurrence relation:

$$\begin{aligned}
F(n) &= bn + c + dF(n/p) = bn + c + d(bn/p + c + dF(n/p^2)) \\
&= bn + bnd/p + c + cd + d^2(bn/p^2 + c + dF(n/p^3)) \\
&= bn + bnd/p + bn(d/p)^2 + c + cd + cd^2 + d^3F(n/p^3) \\
&= bn((\frac{d}{p})^0 + (\frac{d}{p})^1 + (\frac{d}{p})^2 + \ldots + (\frac{d}{p})^{K-1}) \quad \mathbf{<= Let}\ \boldsymbol{n = p^K} \\
&+ c(d^0 + d^1 + d^2 + \ldots + d^{K-1}) + d^K F(\frac{n}{p^K}) \\
&= bn\frac{1-(\frac{d}{p})^K}{1-\frac{d}{p}} + c\frac{1-d^K}{1-d} + d^K a
\end{aligned}$$

Recurrence Relation: Class 3

Continuing:

$$F(n) = bn\frac{1-(\frac{d}{p})^k}{1-\frac{d}{p}} + c\frac{1-d^K}{1-d} + d^K a$$

Recurrence Relation: Class 3

Continuing:

$$F(n) = bn\frac{1-(\frac{d}{p})^k}{1-\frac{d}{p}} + c\frac{1-d^K}{1-d} + d^K a$$

$$= \frac{bnp}{p-d} - \frac{bnp}{p-d}(\frac{d}{p})^K + \frac{c}{1-d} - \frac{cd^K}{1-d} + d^K a$$

Recurrence Relation: Class 3

Continuing:

$$
\begin{aligned}
F(n) &= bn\frac{1-(\frac{d}{p})^k}{1-\frac{d}{p}} + c\frac{1-d^K}{1-d} + d^K a \\
&= \frac{bnp}{p-d} - \frac{bnp}{p-d}(\frac{d}{p})^K + \frac{c}{1-d} - \frac{cd^K}{1-d} + d^K a \\
&= (a + \frac{bp}{d-p} + \frac{c}{d-1})d^K - \frac{c}{d-1} - \frac{bp}{d-p}n
\end{aligned}
$$

Recurrence Relation: Class 3

Continuing:

$$
\begin{aligned}
F(n) &= bn\frac{1-(\frac{d}{p})^k}{1-\frac{d}{p}} + c\frac{1-d^K}{1-d} + d^K a \\
&= \frac{bnp}{p-d} - \frac{bnp}{p-d}(\frac{d}{p})^K + \frac{c}{1-d} - \frac{cd^K}{1-d} + d^K a \\
&= (a + \frac{bp}{d-p} + \frac{c}{d-1})d^K - \frac{c}{d-1} - \frac{bp}{d-p}n \\
F(n) &= (a + \frac{bp}{d-p} + \frac{c}{d-1})n^{\log_p d} - \frac{c}{d-1} - \frac{bp}{d-p}n
\end{aligned}
$$

$$
\begin{aligned}
\log_p n &= \frac{\log n}{\log p}\frac{\log d}{\log d} = \log_d n \frac{\log d}{\log p} = \log_d n \log_p d \\
&= d^{\log_d n \ \log_p d} = n^{\log_p d}
\end{aligned}
$$

Summary of General Recurrence Relations

- Class 1:

$$F(1) = a$$
$$F(n) = b + cF(n-1)$$
$$F(n) = c^n\left(\frac{a}{c} - \left(\frac{b}{c} + \frac{b}{1-c}\right)\right) - \frac{b}{c-1}$$

- Class 2:

$$F(1) = a$$
$$F(n) = bn + c + dF(n-1)$$
$$F(n) = \frac{bn}{1-d} + \frac{c-d(b+c)}{(1-d)^2} + \left(\frac{b}{(1-d)^2} + \frac{a-b-c}{d} + \frac{b+c}{d-1}\right)d^n$$

- Class 3:

$$F(1) = a$$
$$F(n) = bn + c + dF(n/p)$$
$$F(n) = \left(a + \frac{bp}{d-p} + \frac{c}{d-1}\right)n^{\log_p d} - \frac{c}{d-1} - \frac{bp}{d-p}n$$

Table of Contents

How Fast can Algorithms be Solved?

- We use **deterministic** computers
- Deterministic computer makes one exactly determined choice at each choice point
  - Given n alternatives, a deterministic computer selects exactly one alternative
- Time required for a polynomial (**P**) time algorithm to run on a deterministic computer is $O(n^k)$
- What is a non-deterministic polynomial **NP** time algorithm?
- A **non-deterministic** computer should be able to solve an **NP** algorithm in polynomial time
  - A non-deterministic computer has magical powers that can select amongst many alternatives the correct alternative that leads to the right choice
  - There is no need to back-track!

The Biggest Mystery in Computer Science:

- Is NP=P?
- **NP-Complete** problems are problems that can be solved in $O(n^k)$ time on a non-deterministic machine
- It is possible to translate NP problems to NP-complete problems in polynomial time
- The fastest time that an NP-complete problem can be solved is exponential
- If it is possible to solve any NP problem in polynomial time, all NP problems can be solved in polynomial time!
- So far no solid proof has not been established to show whether NP=P or NP$\neq$P
- If you figure this out, you will become very rich!!!