BITS Pilani
Pilani|Dubai|Goa|Hyderabad

BITS Pilani presentation

Dr S. S. Chauhan
sansar@wilp.bits-pilani.ac.in

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Data Structures and Algorithms Design

Lecture No. 3

Dr S. S. Chauhan

Contents

innovate achieve lead

1. Analyzing Recursive Algorithms:
   a) Solving recurrence equations.
   b) Master Theorem.
2. Case Study: Analyzing Algorithms
3. Stack: Stack ADT and Implementation.
4. Queues: Queue ADT and Implementation,
5. List ADT and Implementation.

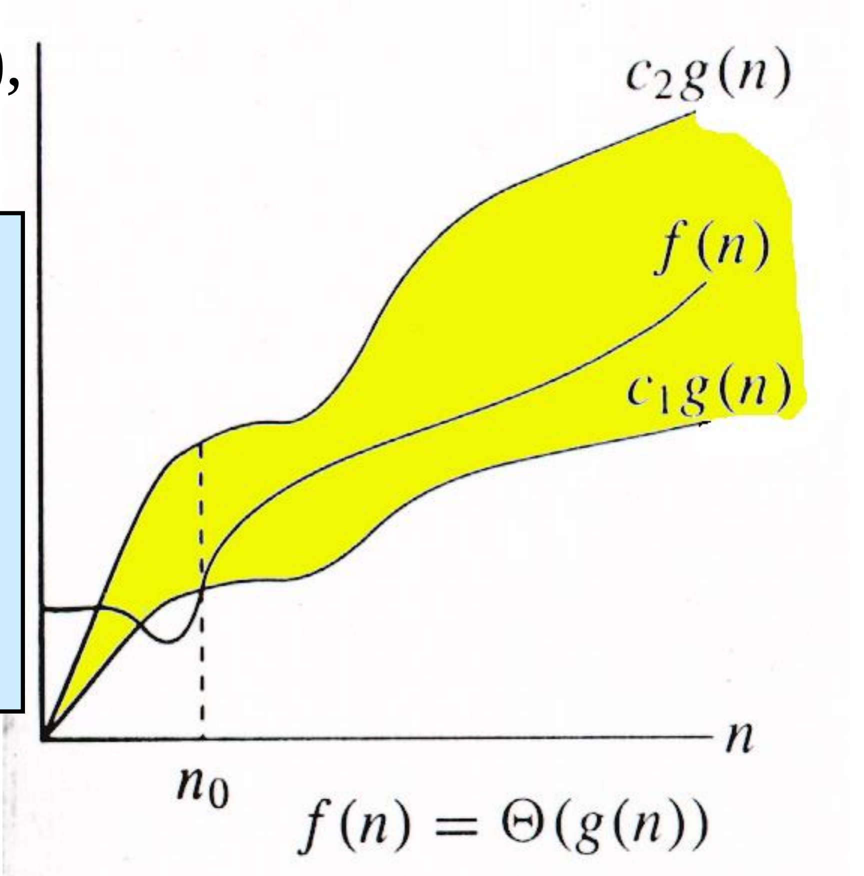
Asymptotic Notation (Revision)

a) Big-Oh
b) Omega
c) Theta
d) Little-Oh, and
e) Little-Omega Notation

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n):$
$\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\}$



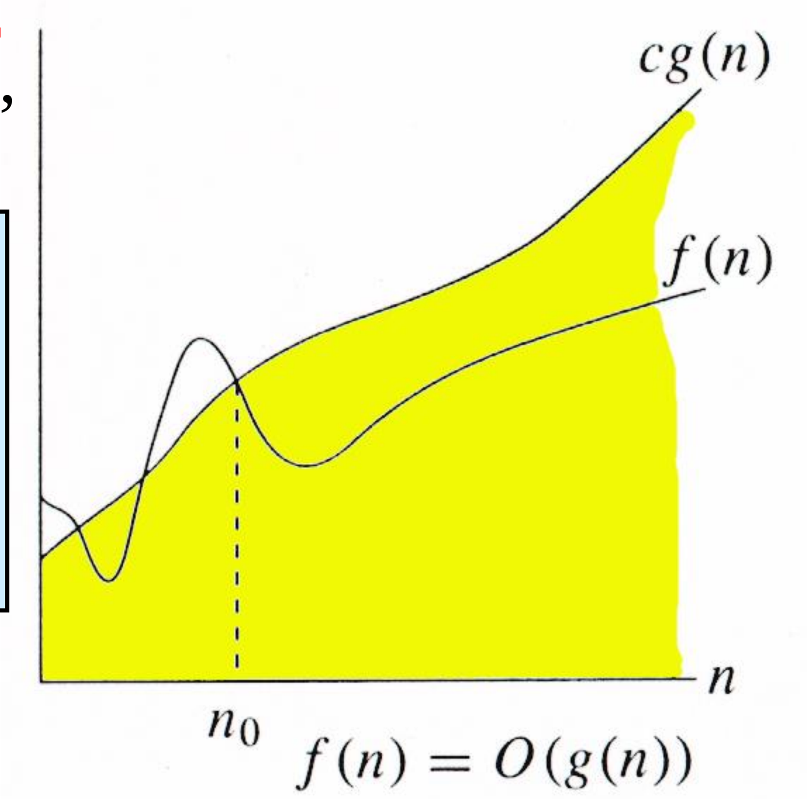
Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

***g*(*n*) is an *asymptotically tight bound* for *f*(*n*).**

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{f(n):$
$\exists$ positive constants c and n_0,
such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq cg(n)\}$



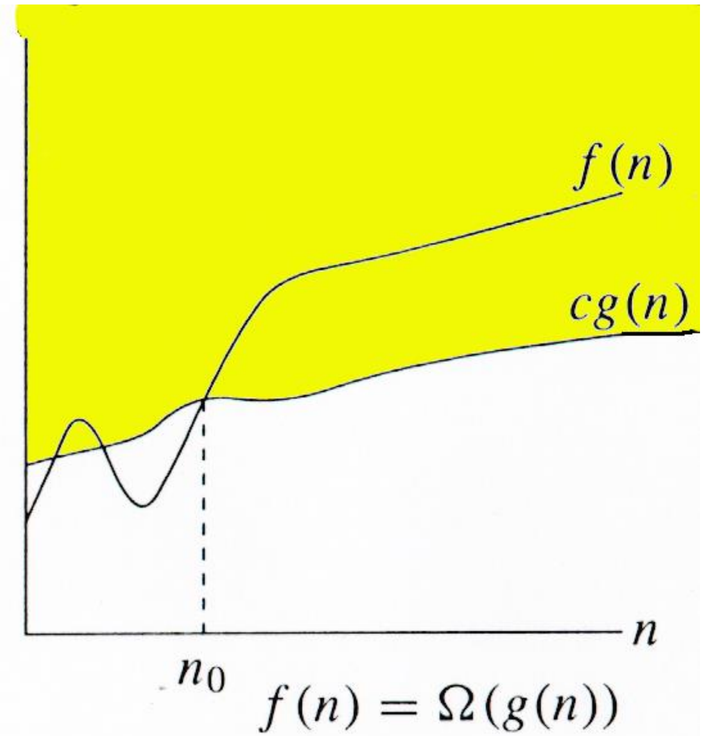
Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

$g(n)$ is an *asymptotic upper bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$.
$\Theta(g(n)) \subset O(g(n))$.

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:



$\Omega(g(n)) = \{f(n) :$
$\exists$ **positive constants** c **and** n_0,
such that $\forall n \geq n_0$,
we have $0 \leq cg(n) \leq f(n)\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

***g*(*n*) is an *asymptotic lower bound* for *f*(*n*).**

$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$.

$\Theta(g(n)) \subset \Omega(g(n))$.

o-notation

For a given function $g(n)$, the set little-o:

$$o(g(n)) = \{f(n): \forall\, c > 0, \exists\, n_0 > 0 \text{ such that } \forall\, n \geq n_0, \text{ we have } 0 \leq f(n) < cg(n)\}.$$

$f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:

$$\lim_{n\to\infty} [f(n) / g(n)] = 0 \qquad \{a/\infty = 0\}$$

$g(n)$ is an ***upper bound*** for $f(n)$ that is not asymptotically tight.

Observe the difference in this definition from previous ones. **Why?**

ω-notation

For a given function $g(n)$, the set little-omega:

$$\omega(g(n)) = \{f(n): \forall\, c > 0, \exists\, n_0 > 0 \text{ such that } \forall\, n \geq n_0, \text{ we have } 0 \leq cg(n) < f(n)\}.$$

$f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity:

$$\lim_{n\to\infty} [f(n) / g(n)] = \infty. \qquad \{\infty/a = \infty\}$$

$g(n)$ is a ***lower bound*** for $f(n)$ that is not asymptotically tight.

Recurrences

- When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence.
- A recurrence is a function defined in terms of:
  - One or more base case, and
  - Itself with smaller arguments
  - Ex

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1) + 1 & \text{if } n>1 \end{cases}$$

- Solution: T(n)=O(n)

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + \mathrm{n} & \text{if } n>1 \end{cases}$$

Solution: T(n)=O(nlgn)

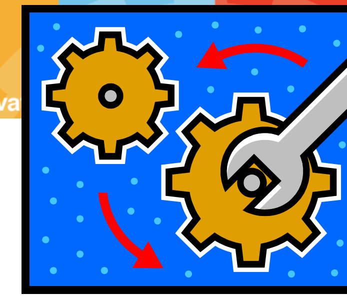
$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/3) + \mathrm{T(2n/3)+n} & \text{if } n>1 \end{cases}$$

Solution: T(n)=O(nlgn)

Solving recurrence equations.

- The Iterative Substitution Method
- The Recursion Tree
- The Guess-and-Test Method
- The Master Method

Iterative Substitution



In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&= 2(2T(n/2^2)) + b(n/2)) + bn \\
&= 2^2 T(n/2^2) + 2bn \\
&= 2^3 T(n/2^3) + 3bn \\
&= 2^4 T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^i T(n/2^i) + ibn
\end{aligned}$$

Note that base, T(n)=b, case occurs when 2^i=n. That is, i = log n.

So,

$$T(n) = bn + bn \log n$$

Thus, T(n) is O(n log n).

Solving T(n) = 3T(n-2) with iterative method

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

My first step was to iteratively substitute terms to arrive at a general form:

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

$=3T(n-4)$

$T(n)=3*3T(n-4)$

Leading to the general form:

$T(n)=3^k T(n\text{-}2k)$

$n-2k=1$ for k, which is the point where the recurrence stops (where T(1)) and

Insert that value $(n/2-1/2=k)$ into the general form:

$T(n)=3^{n/2\text{-}1/2}$

$$= \frac{1}{\sqrt{3}}(\sqrt{3})^n \qquad T(n) = O(3^{n/2}) = O(\sqrt{3^n}).$$

Recursion Tree Method to Solve Recurrence Relations

Recursion Tree is another method for solving the recurrence relations.

A recursion tree is a tree where each node represents the cost of a certain recursive sub-problem.

We sum up the values in each node to get the cost of the entire algorithm.

Steps in Recursion Tree Method to Solve Recurrence Relations

Step-01:

Draw a recursion tree based on the given recurrence relation.

Step-02:

Determine-

- Cost of each level
- Total number of levels in the recursion tree
- Number of nodes in the last level
- Cost of the last level

Step-03:

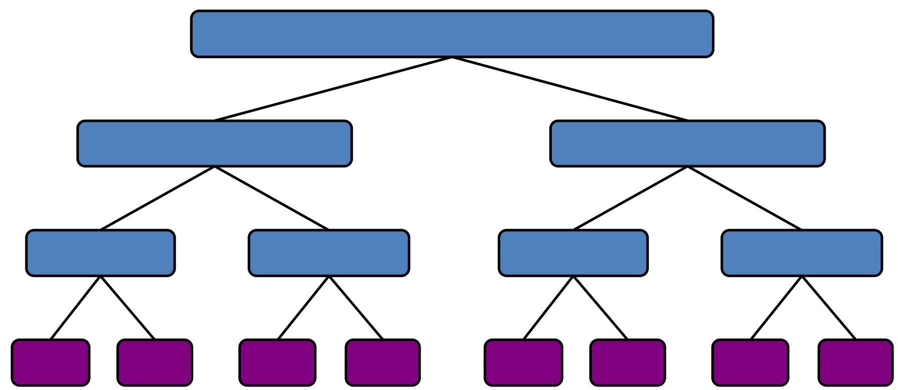
Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation.

The Recursion Tree

Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…



Total time = $bn + bn \log n$

(last level plus all previous levels)

Recursion-Tree Method

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

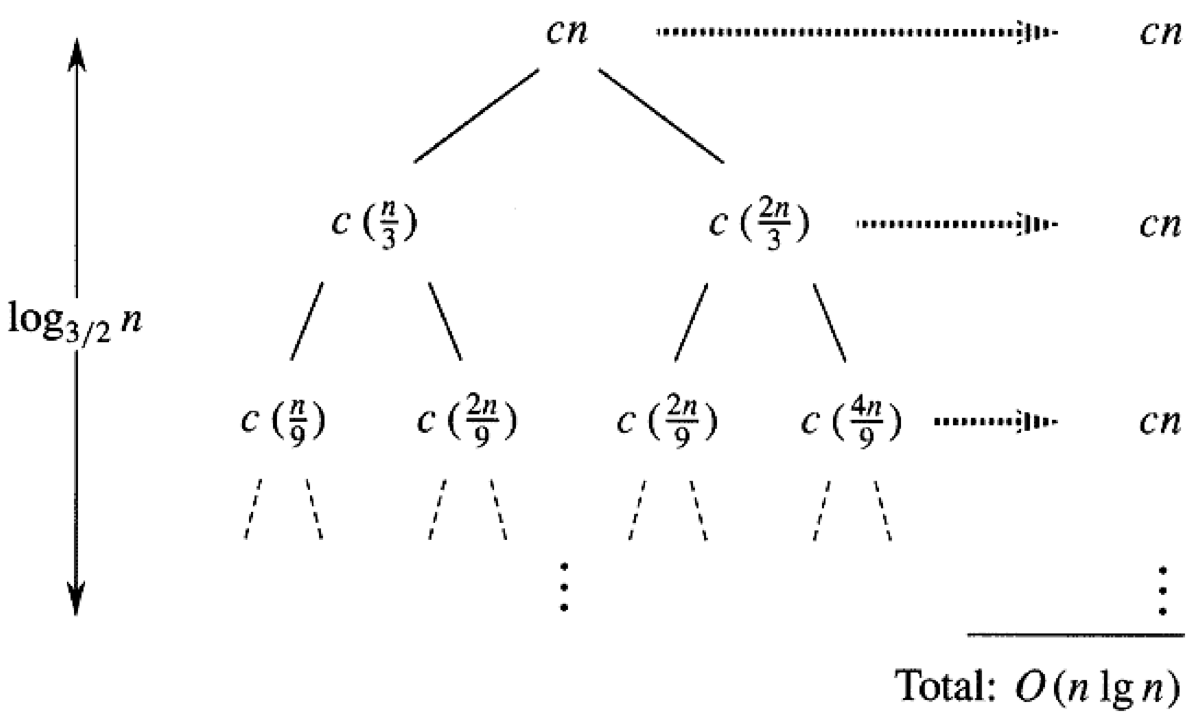
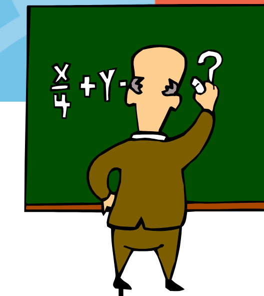


Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.



Guess-and-Test Method

In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

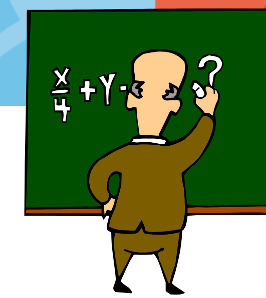
Guess: T(n) < cn log n.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n \end{aligned}$$

Wrong: we cannot make this last line be less than cn log n

* Last line must be less than cnlogn (we want T(n)<=cnlogn, and last line is value of T(n))

Guess-and-Test Method, Part 2



Recall the recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

Guess #2: T(n) < cn $\log^2$ n.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &\leq 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log^2 n - 2\log n + 1) + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \end{aligned}$$

Last line can only be less than equal to cn $\log^2$ n if c > b.

So, T(n) is O(n $\log^2$ n).

In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

Master Method

Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Master Method, Example 1

The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Example:

$$T(n) = 4T(n/2) + n$$

Solution: $\log_b a=2$, so case 1 says T(n) is $O(n^2)$.

Master Method, Example 2

The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

Example:

$$T(n) = 2T(n/2) + n \log n$$

Solution: $\log_b a=1$, so case 2 says T(n) is O(n $\log^2$ n).

Master Method, Example 4

The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

   provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

Example:

$$T(n) = 8T(n/2) + n^2$$

Solution: $\log_b a=3$, so case 1 says T(n) is $O(n^3)$.

Master Method, Example 5

The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Example:

$$T(n) = 9T(n/3) + n^3$$

Solution: $\log_b a=2$, so case 3 where δ =1/3, says T(n) is $O(n^3)$.

Master Method, Example 6

The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

   provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Example:

$$T(n) = T(n/2) + 1$$ (binary search)

Solution: $\log_b a=0$, so case 2 says T(n) is O(log n).

Changing variables

- Examble: Consider the recurrence
$$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n\ ,$$
- Which looks difficult. We can simplify this recurrence, though, with a change of variables. For convenience, we shall not worry about rounding off values, such as $\sqrt{n}$, to be integers.
- Renaming $m = \log n$ $\{n=2^m\}$yields
- $T(2^m) = 2T(2^m/2) + m.$
- We can now rename $S(m) = T(2^m)$ to produce the new recurrence
- $S(\mathrm{m}) = 2S(m/2) + m,$
- which has the solution: $S(m) = O(m \lg m)$.
- Changing back from $S(m)$ to $T(n)$, we obtain $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

Case Studies in Algorithm Analysis

- We show how to use the big-Oh notation to analyze two algorithms that solve the same problem but have different running times.
- The problem we focus on in this section is the one of computing the so-called prefix averages of a sequence of numbers.
- Namely, given an array X storing n numbers,
- we want to compute an array A such that A[i] is the average of elements

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Quadratic-Time Prefix Averages Algorithm

Algorithm prefixAverages1(X):

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0], \ldots, X[i]$.

Let A be an array of n numbers.
for $i \leftarrow 0$ **to** $n-1$ **do**
  $a \leftarrow 0$
  for $j \leftarrow 0$ **to** i **do**
    $a \leftarrow a + X[j]$
  $A[i] \leftarrow a/(i+1)$
return array A

Analysis next slide

Let us analyze the prefixAverages1 algorithm.

- Initializing and returning array A at the beginning and end can be done with a constant number of primitive operations per element, and takes 0(n) time.
- There are two nested for loops, which are controlled by counters i and j, respectively.
- The body of the outer loop, controlled by counter i, is executed n times, for i = 0.....,n - 1.
- Thus, statements a = O and A[i] = a/(i+ 1) are executed n times each.
- This implies that these two statements, plus the incrementing and testing of counter i, contribute a number of primitive operations. proportional to n, that is,0(n) time.

- The body of the inner loop, which is controlled by counter j, is executed i +1 times, depending on the current value of the outer loop counter i.
- Thus, statement a = a + X [j] in the inner ioop is executed 1+2+3+ ....+ n times. (We know that 1+2+3 +« +n = n(n+ 1))
- Thus, the statement in the inner loop contributes $0(n^2)$ time.
- A similar incrementing and testing counter j, also take $0(n^2)$ time
- The running time Of algorithm prefixAverages1 is given by the sum of three terms.
- The first and the second term are 0(n), and the third term is $0(n^2)$.
- So the running time of prefixAveragesi is $0(n^2)$.

In order to compute prefix averages more efficiently, we can observe that two consecutive averages A [i - 1] ad A [i] are similar:

$$
\begin{aligned}
A[i-1] &= (X[0]+X[1]+\cdots+X[i-1])/i \\
A[i] &= (X[0]+X[1]+\cdots+X[i-1]+X[i])/(i+1).
\end{aligned}
$$

If we denote with S the prefix sum X[0]+X[1] + ... +X[i], we can compute the prefix averages as A[i] = S/(i + 1). It is easy to keep track of the current prefix sum while scanning array X with a loop.

A Linear-Time Prefix AveragesAlgorithm

Algorithm prefixAverages2(X):

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that $A[i]$ is the average of elements $X[0], \ldots, X[i]$.

Let A be an array of n numbers.

$s \leftarrow 0$

for $i \leftarrow 0$ **to** $n-1$ **do**

  $s \leftarrow s + X[i]$

  $A[i] \leftarrow s/(i+1)$

return array A

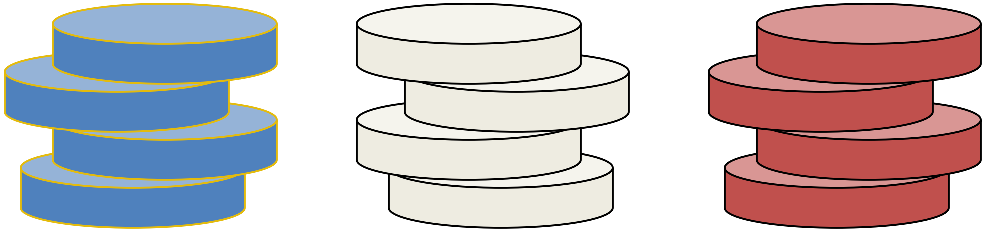
The analysis, of the running time of algorithm prefixAverages2 follows:

- Initializing and returning array A at the beginning and end can be done with a constant number of primitive operations per element, and takes 0(n) time.
- Initializing variable s at the beginning takes 0(1) time
- There is a single for loop, which is controlled by counter i.
- The body of the loop is executed n times, for i = 0,... ,n-1. Thus, statements s = s +X[i] and A [i] = s/ (i + 1) are executed n times each.
- This implies 'that 'these two statements plus the incrementing and testing of counter z continue a number of primitive operations proportional to n, that is, 0(n) time.

Summary

- The running time of algorithm prefixAverages2 is given by the sum of three terms.
- The first and the third term are 0(n), and the second term is 0(1).
- Thus the running time of prefixAverages2 is 0(n), which is much better than the quadratic-time algorithm prefixAverages1.

Stacks



Abstract Data Types (ADTs)

- Abstract data type (ADT) are a way of classifying **data structures** based on how they are used and the behaviors they provide. They do not specify how the **data structure** must be implemented but simply provide a minimal expected interface and set of behaviors
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

Example: ADT modeling a simple stock trading system

- The data stored are buy/sell orders
- The operations supported are
  - order buy(stock, shares, price)
  - order sell(stock, shares, price)
  - void cancel(order)
- Error conditions:
  - Buy/sell a nonexistent stock
  - Cancel a nonexistent order

The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element

Auxiliary stack operations:

- object top(): returns the last inserted element without removing it
- integer size(): returns the number of elements stored
- boolean isEmpty(): indicates whether no elements are stored

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

Applications of Stacks

Direct applications

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of method calls in the Java Virtual Machine

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

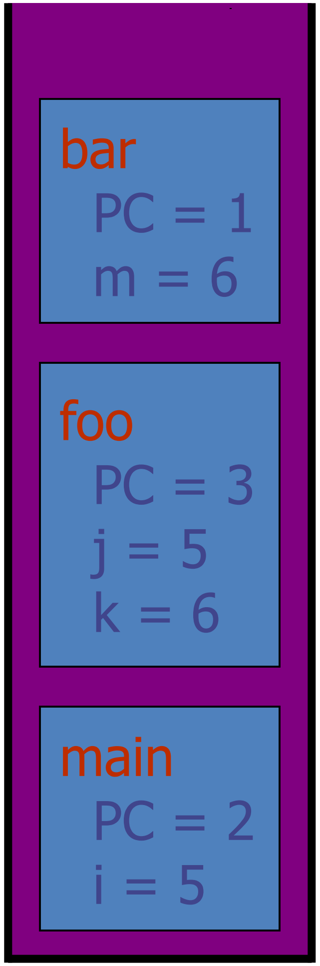
Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  – Local variables and return value
  – Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
}

foo(int j) {
  int k;
  k = j+1;
  bar(k);
}

bar(int m) {
  …
}
```

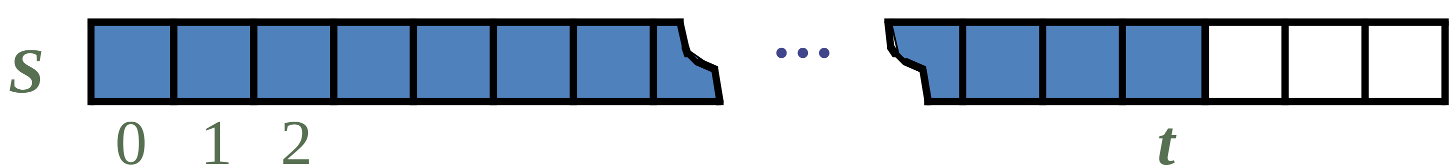


Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Algorithm *size*()
  return $t + 1$

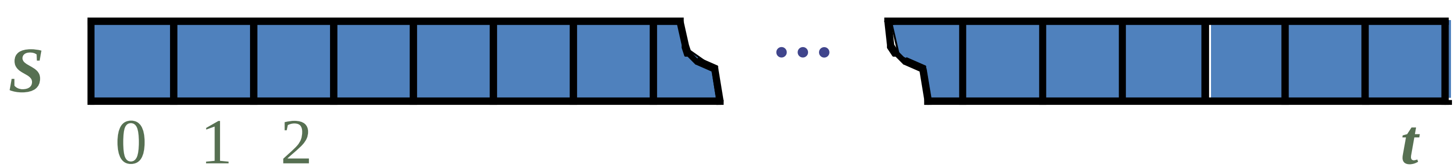
Algorithm *pop*()
  if *isEmpty*() **then**
    throw *EmptyStackException*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm *push*(*o*)
  if $t = S.length - 1$ **then**
    throw *FullStackException*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$



Performance and Limitations

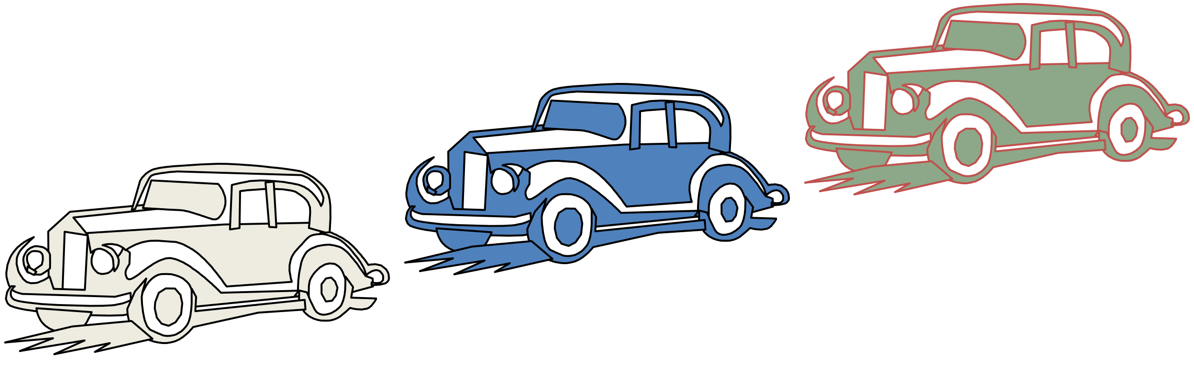
Performance

- Let n be the number of elements in the stack
- The space used is $O(n)$
- Each operation runs in time $O(1)$

Limitations

- The maximum size of the stack must be defined a priori and cannot be changed
- Trying to push a new element into a full stack causes an implementation-specific exception

Queues



The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

Applications of Queues

Direct applications

- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming

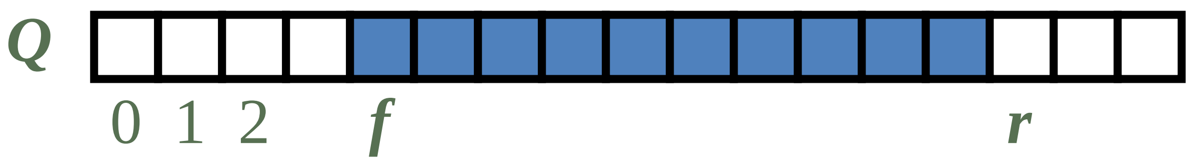
Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

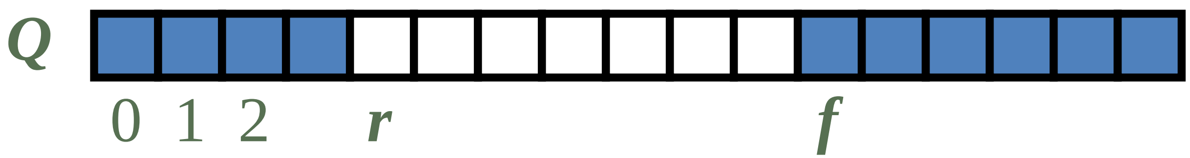
Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

normal configuration



wrapped-around configuration

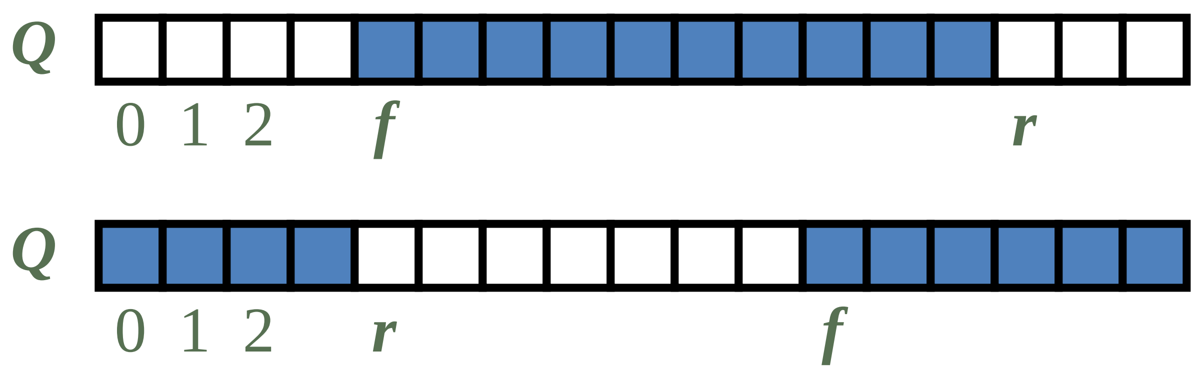


Queue Operations

We use the modulo operator (remainder of division)

Algorithm *size*()
  return $(N - f + r) \bmod N$

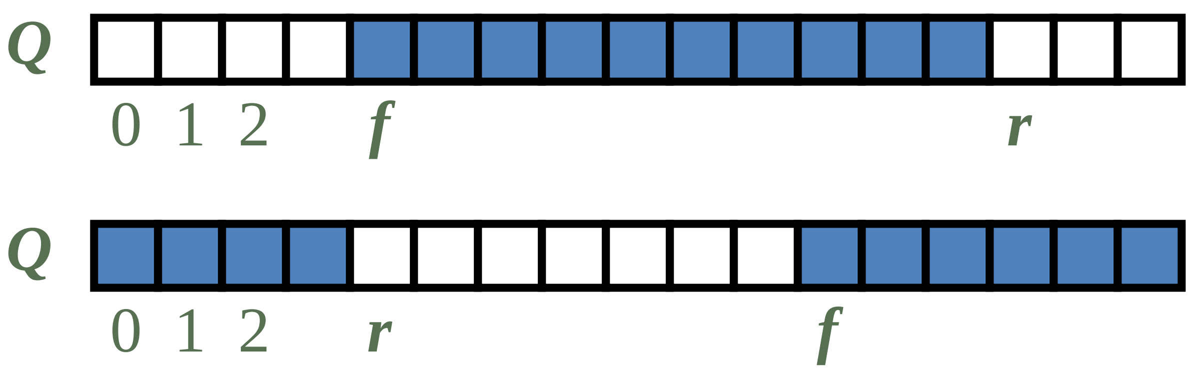
Algorithm *isEmpty*()
  return $(f = r)$



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

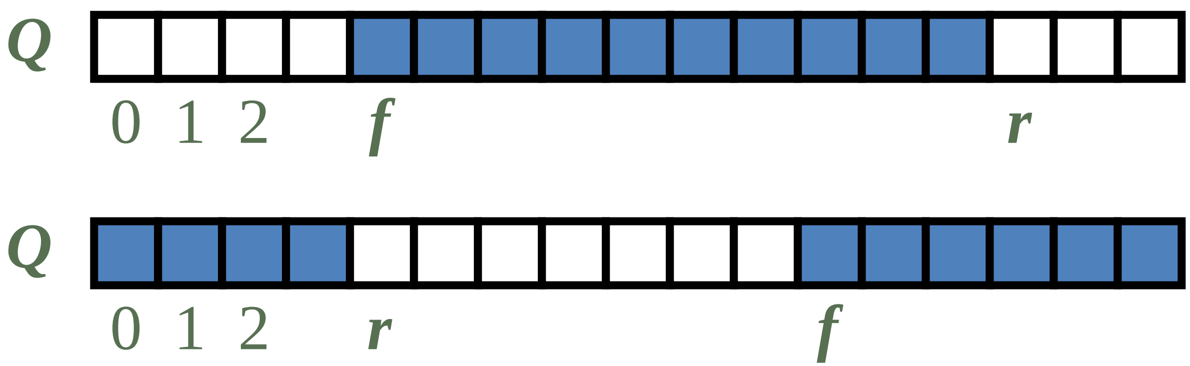
Algorithm *enqueue*(*o*)
  if *size*() = $N - 1$ **then**
    throw *FullQueueException*
  else
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

Algorithm *dequeue*()
  if *isEmpty*() **then**
    throw *EmptyQueueException*
  else
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    return o



Growable Array-based Queue

In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Amortized Analysis

- Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster.
- In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation

Three Methods of Amortized Analysis

- **Aggregate analysis:**
  - Total cost of n operations/n,
- **Accounting method:**
  - Assign each type of operation an (different) amortized cost
  - overcharge some operations,
  - store the overcharge as credit on specific objects,
  - then use the credit for compensation for some later operations.
- **Potential method:**
  - Same as accounting method
  - But store the credit as "potential energy" and as a whole.

Example for amortized analysis

- Stack operations:
  - PUSH(S,x), $O(1)$
  - POP(S), $O(1)$
  - MULTIPOP(S,k), min(s,k)

    while not STACK-EMPTY(S) and k>0

    do POP(S)

    k=k-1

- Let us consider a sequence of n PUSH, n POP, n MULTIPOP.
  - The worst case cost for MULTIPOP in the sequence is $O(n)$, since the stack size is at most n.
  - thus the cost of the sequence is $O(n^2)$. Correct, but not tight.

Aggregate Analysis

- In fact, a sequence of n operations on an initially empty stack cost at most $O(n)$.
- Each object can be POP only once (including in MULTIPOP) for each time it is PUSHed.
  #POPs is at most #PUSHs, which is at most n.
- Thus the average cost of an operation is $O(n)/n = O(1)$.
- Amortized cost in aggregate analysis is defined to be average cost.

Amortized Analysis: Accounting Method

- Idea:
  - Assign differing charges to different operations.
  - The amount of the charge is called amortized cost.
  - amortized cost is more or less than actual cost.
  - When amortized cost > actual cost, the difference is saved in specific objects as credits.
  - The credits can be used by later operations whose amortized cost < actual cost.
- As a comparison, in aggregate analysis, all operations have same amortized costs.

Accounting Method (cont.)

- Conditions:
  - suppose actual cost is c_i for the ith operation in the sequence, and amortized cost is c_i',
  - $\sum_{i=1}^{n} c_i' \geq \sum_{i=1}^{n} c_i$ should hold.
    - since we want to show the average cost per operation is small using amortized cost, we need the total amortized cost is an upper bound of total actual cost.
    - holds for all sequences of operations.
  - Total credits is $\sum_{i=1}^{n} c_i' - \sum_{i=1}^{n} c_i$, which should be nonnegative,
    - Moreover, $\sum_{i=1}^{t} c_i' - \sum_{i=1}^{t} c_i \geq 0$ for any $t>0$.

Accounting Method: Stack Operations

- Actual costs:
  - PUSH :1, POP :1, MULTIPOP: min(s,k).
- Let assign the following amortized costs:
  - PUSH:2, POP: 0, MULTIPOP: 0.
- Similar to a stack of plates in a cafeteria.
  - Suppose $1 represents a unit cost.
  - When pushing a plate, use one dollar to pay the actual cost of the push and leave one dollar on the plate as credit.
  - Whenever POPing a plate, the one dollar on the plate is used to pay the actual cost of the POP. (same for MULTIPOP).
  - By charging PUSH a little more, do not charge POP or MULTIPOP.
- The total amortized cost for n PUSH, n POP, n MULTIPOP is $O(n)$, thus $O(1)$ for average amortized cost for each operation.
- Conditions hold: total amortized cost $\geq$total actual cost, and amount of credits never becomes negative.

The Potential Method

- Same as accounting method: something prepaid is used later.
- Different from accounting method
  - The prepaid work not as credit, but as "potential energy", or "potential".
  - The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

The Potential Method (cont.)

- Initial data structure D_0,
- n operations, resulting in $D_0, D_1, \ldots, D_n$ with costs $c_1, c_2, \ldots, c_n$.
- A potential function Φ: $\{D_i\} \rightarrow$ R (real numbers)

$\forall$ $\Phi(D_i)$ is called the potential of D_i.

- Amortized cost c_i' of the ith operation is:
  - $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$. (actual cost + potential change)

$\forall$ $\sum_{i=1}^{n} c_i' = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$

- $= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$

The Potential Method (cont.)

- If $\Phi(D_n) \geq \Phi(D_0)$, then total amortized cost is an upper bound of total actual cost.
- But we do not know how many operations, so $\Phi(D_i) \geq \Phi(D_0)$ is required for any i.
- It is convenient to define $\Phi(D_0)=0$,and so $\Phi(D_i) \geq 0$, for all i.
- If the potential change is positive (i.e., $\Phi(D_i) - \Phi(D_{i-1})>0$), then c_i' is an overcharge (so store the increase as potential),
- otherwise, undercharge (discharge the potential to pay the actual cost).

Potential method: stack operation

- Potential for a stack is the number of objects in the stack.
- So $\Phi(D_0)=0$, and $\Phi(D_i) \geq 0$
- Amortized cost of stack operations:
  - PUSH:
    - Potential change: $\Phi(D_i)- \Phi(D_{i-1}) =(s+1)-s =1$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})=1+1=2$.
  - POP:
    - Potential change: $\Phi(D_i)- \Phi(D_{i-1}) =(s-1) –s= -1$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})=1+(-1)=0$.
  - MULTIPOP(S,k): k'=min(s,k)
    - Potential change: $\Phi(D_i)- \Phi(D_{i-1}) = –k'$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})=k'+(-k')=0$.
- So amortized cost of each operation is $O(1)$, and total amortized cost of n operations is $O(n)$.
- Since total amortized cost is an upper bound of actual cost, the worse case cost of n operations is $O(n)$.

Solve:

$$T(n) = 9T(n/3) + n.$$

Example 1: $T(n) = 9T(n/3)+n$.

Here $a = 9$, $b = 3$, $f(n) = n$, and $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since

$$f(n) = O(n^{\log_3 9-\epsilon}) \text{ for } \epsilon = 1$$

case 1 of the master theorem applies, and the solution is $T(n) = \Theta(n^2)$.

Solve:

$T(n) = T(2n/3)+1.$

$T(n) = T(2n/3)+1$.

Here $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^0 = 1$.

Since $f(n) = \Theta(n^{\log_b a})$, case 2 of the master theorem applies,

so the solution is $T(n) = \Theta(\log n)$.

$$T(n) = 3T(n/4) + n.$$

$T(n) = 3T(n/4) + n.$

Here a=3, b=4, f(n)=n

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793}).\ \text{For } \epsilon = 0.2$$

we have $f(n) = \Omega(n^{\log_4 3+\epsilon}\ )$

So case 3 applies if we can show that

$af(n/b) \le cf(n)$ for some $c < 1$ and all sufficiently large n.

This would mean 3(n/4)<=cn

Setting $c = 3/4$ would cause this condition to be satisfied.

Solution is T(n)=O(nlogn)

Solve using master method

a. $T(n) = 2T(n/2) + \log n$
b. $T(n) = 8T(n/2) + n^2$
c. $T(n) = 16T(n/2) + (n \log n)^4$
d. $T(n) = 7T(n/3) + n$
e. $T(n) = 9T(n/3) + n^3 \log n$

Solution

innovate achieve lead

1. $T(n) = 2T(n/2) + \log n$

   Case 1. $\Theta(n)$.

2. $T(n) = 8T(n/2) + n^2$.

   Case 1. $\Theta(n^3)$.

3. $T(n) = 16T(n/2) + (n \log n)^4$

   Case 2. $\Theta(n^4 \log^5 n)$.

4. $T(n) = 7T(n/3) + n$

   Case 1. $\Theta(n^{\log_3 7})$

5. $T(n) = 9T(n/3) + n^3 \log n$

   Case 3. $\Theta(n^3 \log n)$.