ALG KOMPENDIUM

Noter til Algoritmer og datastrukturer

30. MAJ 2020

SKREVET AF SIMON HELMING NIELSEN

Table of Contents

Sorting Algorithms

Sorting algorithms all try to solve the sorting problem.

The sorting problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$ of the input such that $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

Insertion Sort

Insertion sort is an Efficient algorithm for sorting a small number of elements.

Its worst-case time is $\Theta(n^2)$ and it uses constant space $\Theta(1)$

The procedure takes an array $A[1..n]$ and sorts its content

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

- $A[1..j-1]$ are the cards in the hand
- $A[j..n]$ are the cards in the table

It works reducing the problem to the smallest possible problem and expanding from there.

We can assume that the first position in the array is sorted in relation to itself.

Then we sort the second position in relation to the first. Then we know the first two is sorted. We just continue through the entire array inserting the next object before the first element that is bigger. An example can be seen in Figure 1

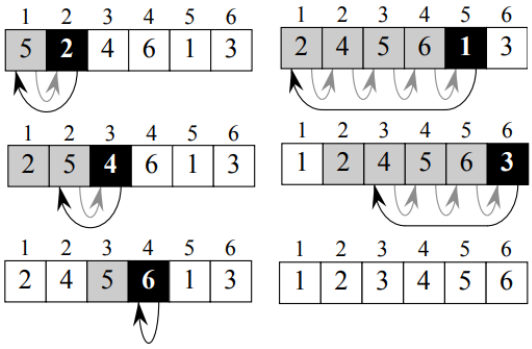


FIGURE 1 INSERTION SORT VISUALIZED

Merge Sort

Merge sort relies on the Divide and conquer method

Its worst-case time is $\Theta(n \log(n))$ and it uses a linear amount of space $\Theta(n)$

It uses an auxiliary function to sort after dividing into trivially solved subproblems

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋          Divide
3      MERGE-SORT(A, p, q)      Conquer
4      MERGE-SORT(A, q + 1, r)  Conquer
5      MERGE(A, p, q, r)        Combine
```

As can pe seen merge sort calls itself on two parts of the array which is split approximately in the this continues until p>=r, this happens when there is only one element in the subarray.

When this happens the merge function is called

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Merge assumes the values between p and q and the ones between q and r is sorted with respect to themselves. These values are then printed into two new array L and R in lines 1 to 9. **Notice that they are both appended by an infinity value**. When this is done, we compare the first element in each array and put that in the first spot at the first iteration that is p. this keeps going until the entire array between p and q has been filled. The infinity values are appended so we know when an array has been emptied. This is done from line 12 to 17.

Examples of this can be seen in the slide for lecture 3

Heap Sort

Heapsort is an algorithm that sorts in place without using auxiliary arrays. Is uses heaps as a data structure to sort an array.

Its worst-case time is $\Theta(n \log(n))$ and it uses a constant amount of space $\Theta(1)$

The main idea

1. Build a Max-Heap from $A[1..n]$.
2. Take the max element $A[1]$ and swap it with $A[n]$ (which is a leaf node!)
3. Restore the max-heap property for $A[1..n-1]$
4. Repeat steps 2—4 on the subarray $A[1..n-1]$

We set the highest value to be in the last position n and restore heap property for the rest of this array. We then store the biggest number in n-1 and so on and so on.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Order matters!

Decreasing *heap-size* = focus on the tree representation of the subarray $A[1..heap\text{-}size]$

At line 5 the subtrees rooted at LEFT(1) and RIGHT(1) are max-heaps

We can go down to 2 since the last element is trivially sorted.

We reduce the heap size to makes sure the last element (where we stored the biggest numer) is not moved .

The subfunction build-max-heap and max-heapify can be found in Heaps from binary trees section.

We can simply use max-heapify to restore the properties since the subtrees of the root node should still be max heaps (when not taking n into account)

Running time:

The running time is calculated from the run times of BUILD-MAX-HEAPIFY and MAX-HEAPIFY. BUILD-MAX-HEAP takes Θ(n) and MAX-HEAPIFY takes O(log(n)). MAX-HEAPIFY is run n-1 times because of the for loop in line 2 so the running time is:

$$O(n\log(n)) + \Theta(\mathrm{n}) = O(n\log(n))$$

Quicksort

Quicksort is an algorithm that is based on divide and conquer and works in constant time.

Its worst-case running time is $O(n^2)$ and average is $\Theta(n\log(n))$ **it is worth noting the constant factors hidden behind the asymptotic notation is usually low, so it is often a practical choice.**

The main idea

- **Divide:** rearrange the array into two partitions (subarrays) $L = A[p\,..\,q-1]$ and $R = A[q+1..r]$ such that
  - Each element of L is less than or equal to $A[q]$
  - Each element of R is greater than or equal to $A[q]$
- **Conquer:** sort L and R by recursive calls to Quicksort.
- **Combine:** the two subarrays are already sorted, thus no work is needed to combine them.

Quicksort works by partitioning the array into smaller subarrays where one array includes all elements smaller than the element between the subarrays, and the other all elements bigger.

It uses an auxiliary function PARTITION

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Set the pivot element

At the beginning both partitions are empty: separator out of the range $[p, r]$

This partitions the array into the two subarrays mentioned above. It is done in relation to the last element in the array.

It starts from the left of the subarray, and, whenever a value is smaller, I is incremented, and the element smaller than x is put into the I'th spot in the array. This continues until we ran through the entire array (except x) and we exchange the value in i+1 (a value bigger than x) with x. the array is then split. And we return the position of the pivot element.

Runtime

It runs in linear time since the only element that is not constant time is our for loop with runs through n-1 elements.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

Recall that quicksort works by making smaller and smaller partitions in relation to an element.

We can see if p<r we run the algorithm. This means the subarray to sort is bigger than 1

First, we partition the array, and return the pivot element, we then call quicksort on the two subarrays gained from the partition. This keeps going until we end up with arrays with the size of 1, which is trivially sorted. When all subarrays of one is placed correctly according tot heir pivot the array is sorted and we are done.

Runtime:

$$T(n) = T(q-p) + T(r-q) + \Theta(n) \qquad (n = r-p+1)$$
$$= T(m) + T(n-m-1) + \Theta(n) \qquad \text{(change of variable } m = q-p\text{)}$$

Depends on how balanced is the partitioning at each step, which in turns depends on the values of the pivots

Worst case:

Worst case occurs when the partition only returns one subarray (this is when the element x is already the biggest or the smallest element in the array) when this happens at every partition call, we have our worst case. **This occurs when the array is already sorted**

$$\begin{aligned} T(n) &= T(m) + T(n-m-1) + \Theta(n) \\ &= T(n-1) + T(0) + \Theta(n) && (m = n-1) \\ &= T(n-1) + \Theta(n) && (T(0) = \Theta(1)) \\ &= \Theta\left(\sum_{i=1}^{n} i\right) \\ &= \Theta(n^2) && \left(\sum_{i=1}^{n} i = n(n+1)/2\right) \end{aligned}$$

Best case:

This occurs when the partition returns two balanced arrays in all calls.

$$\begin{aligned} T(n) &= T(m) + T(n-m-1) + \Theta(n) \\ &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil - 1) + \Theta(n) && (m = \lfloor n/2 \rfloor) \\ &\cong 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

This also happens in most cases where we do not hit worst case.

Counting Sort

Counting sort is a sorting algorithm that does not use comparisons.

It sorts in linear time $\Theta(n)$ but uses linear extra space $\Theta(n)$

It is stable, i.e., numbers with the same value appear in the same order as they do in the resulting array

The main idea

- Exploit the fact that numbers are integers in the range $0..k$
- Count occurrences of each number in A
- Determine the final (sorted) position of each element, and copy them in the array B

A is the array to sort and k is the largest value allowed in the array and B is the output array

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Counting sort works by counting number of occurrences of each number

It does this by making a new array the size of k+1 (0 to k) and setting all values to 0.

It then runs through the entire Array A and ticks up the counter once per number. (if a 15 is found it ticks up position 15 in array C by one.)

We then, starting from the first element add the preceding element to each element so C instead has the amount of values that is i or less.

We then loop over the length of A and input the value stored in A at the j'th position into its correct position. That is defined by the number corresponding to the A[j]'th position. Afterwards we tick down this value in c with one, so the next occurrence gets places just before it in the array

An example can be seen in lecture 5

Runtime:

The first and 3rd loop takes $\Theta(k)$ time while the second and 4th loop takes $\Theta(n)$

This gives a total runtime of $\Theta(n + \mathrm{k})$

In practice counting sort is only used when k=O(n) and in this case the run time is $\Theta(n)$

Radix Sort

Radix sort is a non-comparative sort function that instead distributes them after their radix.

It is a stable algorithm that runs in linear time.

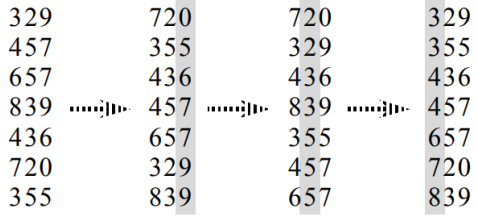
For radix sort to be optimal, all numbers must have the same amount of digits

The main idea

- Repeatedly sort the elements according to the i-th digit
- Start from the least significant digit up to the most significant one.
- It is crucial to use a stable sorting subroutine

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

It is important that the sorting algorithm is stable, so we do not shuffle the sorting in the subsequent sorts.



Runtime:

Since radix takes n sorts, it depends on the sorting subroutine. So, if we use counting sort, we end up with $\boldsymbol{\Theta(d(n+k))}$ typically, **d is constant** and **k=O(n)** in which case the runtime is $\boldsymbol{\Theta(n)}$

Search Algorithms

Binary Search

Binary search is a recursive search algorithm that uses the divide and conquer method.

It solves the element search problem

The element search problem:
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ sorted in non-deceasing order, two indices l, r such that $1 \leq l \leq r \leq n$, and a number a to search
Output: An index $l \leq i \leq r$ such that $a = a_i$ if there exists such an index, 0 otherwise.

```
BIN-SEARCH(A, l, r, a)
 1  if l < r
 2       m = ⌊(l + r)/2⌋
 3       if A[m] ≥ a
 4            return BIN-SEARCH(A, l, m, a)
 5       else
 6            return BIN-SEARCH(A, m + 1, r, a)
 7  elseif l = r and A[l] = a
 8       return l
 9  else
10       return 0
```

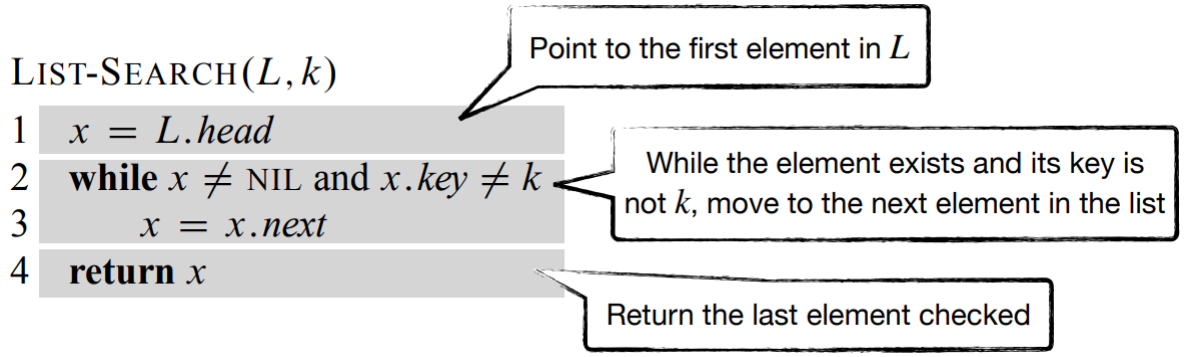
A is the array, L the start of the array to search, r the end of the array to search and a is what we are looking for

This algorithm **Assumes that the array is sorted in non-decreasing order**. This is important because the algorithm uses this assumption to use fewer operations.

Binary search looks at the middle element of the array and depending on if this is bigger or smaller than the element we are looking for, either calls itself on the left or right half of the array. This continues until the array's length is equal to 1 (l=r). when this happens, we return the position if it is found, otherwise returns 0

Linked List Search

The procedure List-Search takes two parameters, the list to search and what to search for.



It simply looks through each element linearly until it is either at the end (x=nil) or the element is found (x.key=k)

It then returns the value or nil if it isn't present.

Runtime:

Since it simply looks at each element the procedure takes Linear time Θ(n) since it may have to scan the entire list.

Data Structures

Stacks

Stacks are a data structure that conforms to the first in, last out policy. This can be visualized as a stack of cards. The first card you put down, is the last to be removed.

Stacks use three methods, ***STACK-EMPTY,*** *which checks if the stack is empty along with* ***PUSH*** and ***POP****, which adds, or removes an element to the top respectively.*

Stacks are stores as arrays and can at most store n elements in an array S[1...n]

the array needs a method ***S.top*** *which is the most recent element in the stack. In other words, the stack consists of the elements in the subarray S[1...S.top]* ***When S.top= 0 the stack is empty All operations in a stack takes constant time.***

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

Queues

Queues are a data structure that conforms to the first in first out policy. This can be visualized as a queue of people waiting in line. The one who waited the longest goes first.

Queues have two methods. ***ENQUEUE*** and ***DEQUEUE*** which adds or removes an element to the queue respectively. A queue of at most n-1 elements can be stored using Q[1...n] this is because the porperty Q.tail needs to point to an empty element.

Queues have two elements. Q.tail and Q.head.

Q.head is the oldest element, which is the first element to be removed.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1

DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Q.tail Points to the next available index in the array. This is where we insert elements to.

The queue wraps around from the end of the array to the beginning when full. This can cause overflows and underflows.

Q underflows when one attempts to dequeue when Q is empty.

Q overflows when one attempts to enqueue an element when Q is full.

Q is empty when Q.tail=Q.head

Q is full when Q.tail+1 =Q.head.

Both operations take constant time.

Linked Lists

Linked Lists are a data structure in which the objects are arranged in a linear order. Unlike arrays, the order in the list is determined by a pointer in each object. They provide a simple and flexible representation of dynamic sets, supporting all usual operations such as insertion, deletion, search,

etc. There are two different types. Doubly linked lists, and singly linked lists. Here we will only explain double linked lists as single linked lists is the same, but without the .prev method.

A doubly linked list **L** is an object with an attribute ***key*** and two other pointer attributes. ***next*** and ***prev.***

This means that a given element x:

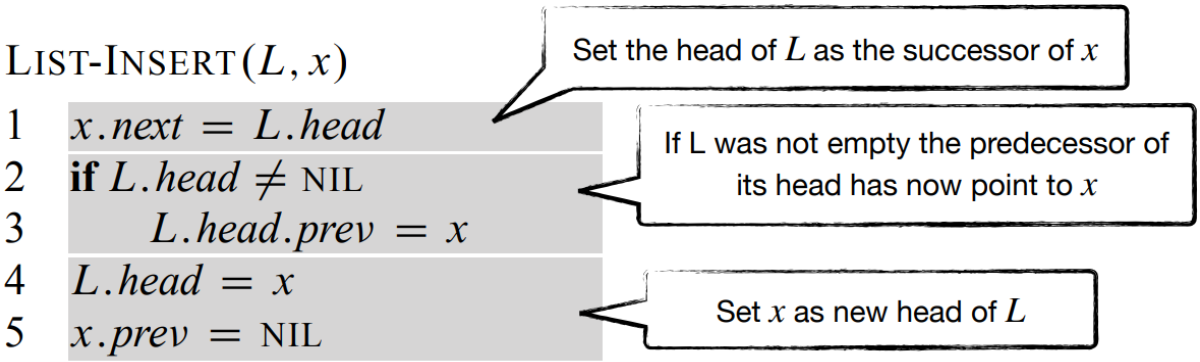
x.next points to the next element in the list (if x.next=nil the object has no successor and is the tail of the list. This means it is end of the list)

x.prev points to its predecessor (if x.prev =nil the object has no predecessor and is the head of the list. This means it is the start of the list.)

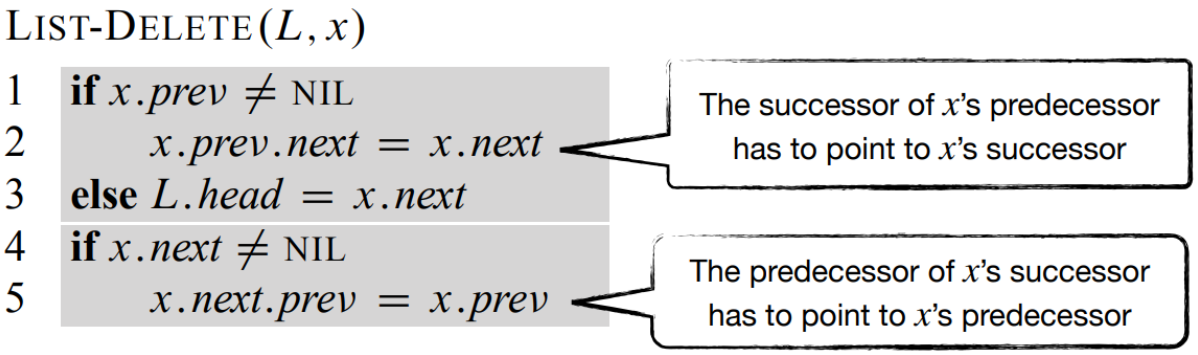
The attribute L.head points to the start of the list. (**Head is always the start**)

How to search a list can be seen in Linked List Search

There are two methods other than search related to linked lists, LIST-INSERT and LIST-DELETE



LIST-INSERT prepends an element to **THE START** of the list. This is done by setting the new element X's next property to L.head. we then check if .head was nil. If it wasn't, we set the prev method of the head to x before change L.head to x. this is to ensure a link to the new element from the head from before insertion.



When Deleting and element we simply set the previous elements next method to x.next. and x.nexts prev method to x.prev.

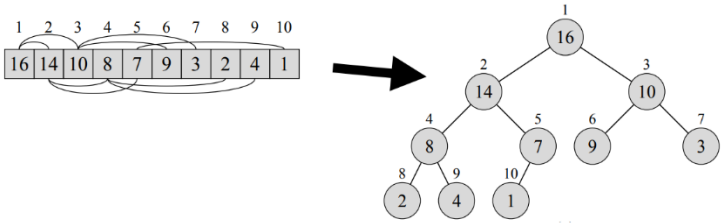
This is to make the elements before and after x point to each other and the list is intact.

Runtime:

Both procedures take constant time Θ(1)

Binary trees

In some cases, it is beneficial to look at arrays as binary trees. this is used in i.e Heapsort



To access certain elements using the principles of the binary tree, we need functions to find the parent node, and left and right sibling.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

Heaps from binary trees

Heaps are a subtype of binary trees. A heap is defined as a binary tree where all children nodes are smaller than their parent nodes (or bigger in case in min heaps). Beneath is explained how to turn a binary tree into a heap.

A heap has two important attributes.

1. length = the number of elements in the array
2. heap-size = number of elements of the heap stored in the array

it is important to note that 0 ≤ heap-size ≤ length

in the following we focus exclusively on max heaps, but these can be modified easily to min heaps.

The property the heap needs to have to be considered a max heap is:

For all nodes i > 0 (i.e., excluding the root) $\mathbf{A[PARENT(i)] \geq A[i]}$

this simply means that the parent should always be bigger than the children.

To generate and use heaps in general, we use two functions.

Max-Heapify: helpful procedure to maintain the max heap property. Runs in O (lg n) time

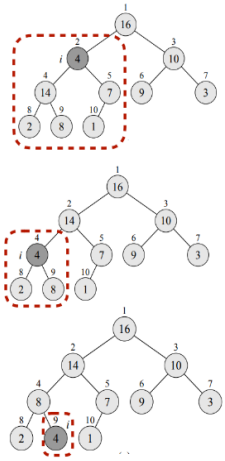
Build-Max-Heap produces a max-heap from an unordered input array. Runs in O(n) time

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Determine the index with the largest value

Swap the values and recursively restore the heap property in the subtree

A is the array to be heapified (recall that an array be a binary tree)



The algorithm assumes both the left and right child is already sorted as heaps and sorts both sub trees and the parent into one heap.

It does this by comparing I with the root of the left and right subtrees and setting whichever is biggest as “largest”.

If i isn’t the largest, we swap the position of i and the largest element., then MAX-HEAPIFY is called recursively on the subtree where i is now the root.

Its worst-case running time can be calculated from the height of the tree. This is because the algorithm runs in constant time and the only factor is the amount of recursive calls, which at most can be the same as the height.

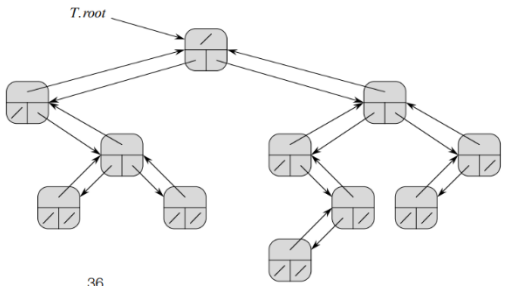
The worst case occurs when the bottom level of the tree is exactly half full, in which case the size of the largest subtree is at most 2n/3. This gives us $T(n) \leq T(2n/3) + \Theta(1)$ which can be solved with the master theorem to be T(n) = O(log n)

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

Max Heap builds a heap from the entirety of A. it does by starting from A.length/2 and building a heap from each element higher in the three (higher in tree is lower I value) we start from this value since this will always be the last node that has a leaf node underneath. We then use heapify to make a small heap out of this one, and we move on to the next node.

Rooted Trees

Rooted trees are a different way of working with Binary trees. This is needed since it is not possible to describe all types of binary trees using the array to binary representation. Instead, it will be structured using pointers. each node in the three will have three properties x.p, x.left and x.right



x.p is the parent node. This is nil if x is the root.

x.left is the left child node and x.right is the right child node. These are nil if none is present.

X may have other attributes to store additional information. Most often is key.

This method can be expanded to trees with more than two children. In this case the methods could simply be named x.child1 x.child2 etc. this would often lead to a waste of memory when many elements do not have the max number of children. There also must be an upper bond for the number of children for this to work. **It is instead suggested to use the Left-child right-sibling scheme.**

Left-child Right-sibling Scheme

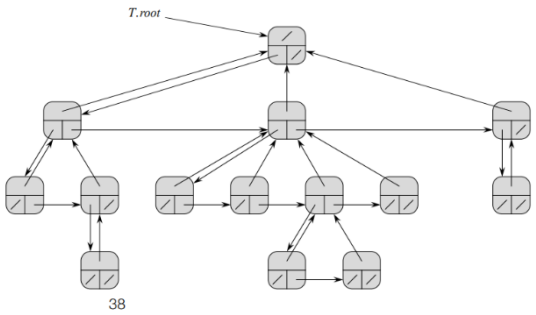
Each node x has three attributes

x.p points to the parent node

x.left-child points to the leftmost child of x

x.right-sibling points to the next sibling of x to the right

x may have other attributes to store additional information. Most often is key.



Binary Search Trees (BST)

Binary search trees are a type of binary trees that support many different operations including Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete. They can both be used as a dictionary and as a priority queue. Many of the basic operations take time proportional to the height of the tree, which for a balanced tree equates to Θ(lg n)

Good idea when the tree is balanced (h = Θ(lg n))

Bad idea when the tree is unbalanced (h = Θ(n))

It can be reasonably assumed that the tree is balanced if only insert is used and the inserted values are in random order. If deletion is used, we cannot say anything (we can say max height is n)

Theorem 12.4

The expected height of a randomly built binary search tree on n distinct keys is $O(\lg n)$.

BSTs can be represented by means of linked data structures where each node has

- A key (and optional satellite data)
- Links to its parent node, left child, and right child

- The root node is the only node in the tree whose parent is Nil
- Keys in the tree satisfy the binary-search-tree property

for the tree to be a valid BST, it must have the following properties:

Binary-search-tree property:

For any node node x in the tree, the following hold

- If y is a node in the left subtree of x then $y.key \leq x.key$
- If z is a node in the right subtree of x then $x.key \leq z.key$

In practice this means, a left child is smaller than the parent, and a right child is bigger than the parent.

A binary search tree can be printed in order by using these properties as done in INORDER-TREE-WALK.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

First recursively visit the left child, then print the node x, and recursively visit the right child

We simply go to the left child until there is no more left children. At this point we have found the lowest value in the tree and we print it, we then check the right child, which would be bigger. This allows us to use the binary search tree to print in order spending **Θ(n**) time since each node is visited exactly once.

Two other tree walks are possible,

- **Preorder tree walk** prints the root before making the recursive calls to the left and right children
- **Post order tree walk** print the root after returning from the recursive calls to the left and right children

Searching in a Binary Search Tree.

The search algorithm can be implemented with an iterative approach or using recursive calls.

It works by exploiting the BST properties. We compare the value at a node with the value of k, if the value is smaller than our k, we go right, if its bigger, we go left, if they are identical or the node is nil, we return the value. **This takes at most O(h) where h is the height of the tree.**

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

We can also find the maximum and the minimum value of the tree by using the properties of BSTs. The leftmost child will always be the smallest element, and the rightmost child will always be the biggest. . **This takes at most O(h) where h is the height of the tree**

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

BST successor/predecessor

We can exploit the properties of a BST to find a successor or predecessor. A successor of x is the node having the smallest key greater than x.key if it exists; otherwise it returns Nil

Only a successor will be described here. A predecessor is found the same way, but with left and right swapped.

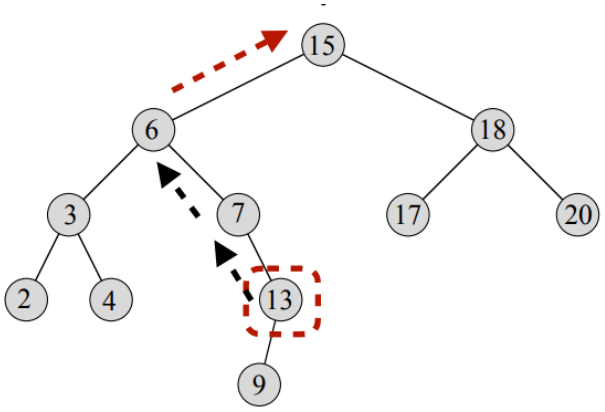
```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

Case 1: minimum element in the right subtree

Case 2: move to the first ancestor who has x in the left subtree, by climbing up the parent pointers

Since we are looking for a bigger value, we first check if there is anything in the right subtree, if there is, we find the minimum value in this subtree.

If this is not possible, we go to the parent, and continue progressing through parent nodes the previous node is no longer the right node of the tree. This case is visualized here. We can easily see, that in both cases, this process takes at most Θ(h) since we only move up or down, but never both.



Inserting in a Binary Search Tree.

The Tree-Insert procedure takes a BST T and a node z and update the links so that z becomes a node of while T preserving the BST property.

```
TREE-INSERT(T, z)
1   y = NIL
2   x = T.root
3   while x ≠ NIL
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == NIL
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

x is the current node and y its parent

Move x and y down in the tree comparing x.key with z.key

At this point x is Nil and z can replace it

Case 1: T was empty. Then, z becomes the root

Case 2: place z in the appropriate subtree by comparing z.key with y.key

We search the tree, checking at each position if the value is bigger or smaller and going left or right depending on this. When a nil node is hit, we are at the right position for our node and it is inserted as a child of y.

Runtime:

The only part not taking constant time is 3-7, and this part takes at most h operations where h is the height of the tree. The run time is Θ(h)

Deleting from a Binary Search Tree.

The procedure for deleting takes a BST T and a node z in t and updates the pointers so that z does no longer belong to t while preserving the BST property.

Basic Idea:

We consider three cases:

1. If z has no children we simply remove it
2. If z has just one child, then we elevate that child to take z's position in the tree.
3. If z has both children, then we find z's successor —which is in z's right subtree and has no left subtree— and have y take z's position in the tree

The pseudocode for Tree-Delete will use a subroutine, called Transplant, that replaces one subtree rooted at node with the subtree rooted at node . **Its worst-case running-time is Θ(1)**

```
TRANSPLANT(T, u, v)
1   if u.p == NIL
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else u.p.right = v
6   if v ≠ NIL
7       v.p = u.p
```

Node u's parent ends up having v as its appropriate child

Node u's parent becomes node v's parent

```
TREE-DELETE(T, z)
1  if z.left == NIL                                   ← Case A
2      TRANSPLANT(T, z, z.right)
3  elseif z.right == NIL                              ← Case B
4      TRANSPLANT(T, z, z.left)
5  else y = TREE-MINIMUM(z.right)                     ← Case C
6      if y.p ≠ z
7          TRANSPLANT(T, y, y.right)
8          y.right = z.right
9          y.right.p = y
10     TRANSPLANT(T, z, y)                            ← Case D
11     y.left = z.left
12     y.left.p = y
```

Case a and b corresponds to 1 and 2 where c and d are two sides of 3.

We first check if z has a child to the left, if not, transplant that child into the position of z.

We then do the same for right. (if both are nil this still works since z is simply replaced with nil)

if z has two children, we find the minimum value in the right subtree. If this is not a direct descendant of z, we transplant y with y.right. we then say that y.right is now z-right ad y.right.p is y. then we can transplant y into z's position and remove z from the tree. Below the result of this operation can be seen on a tree, where z.key = 10

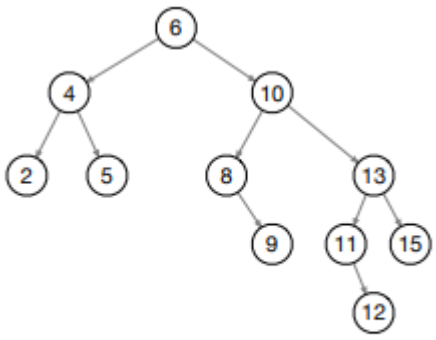
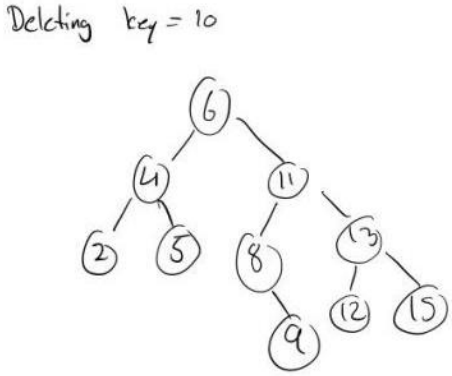


Figure 1: Binary Tree



Runtime:

The only thing that does not take constant time is TREE-MINIMUM which takes Θ(h) time, where h is the height of the tree.

Red-Black Trees (RBT)

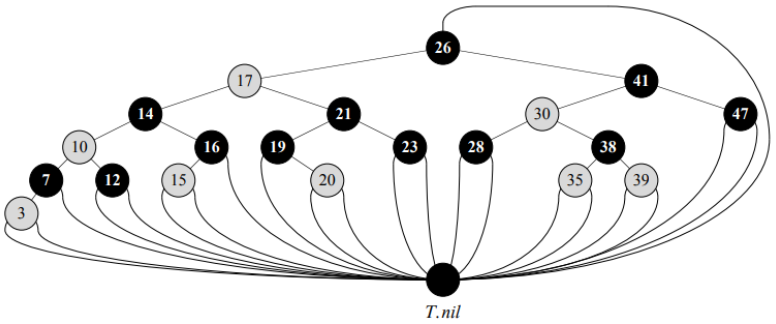
An RBT is a type of BSTs that satisfies the red-black properties. This ensures that the tree is approximately balanced even when deletion is used.

A red-black tree with n internal nodes has height at most $h = 2\log(n+1)$

Red-back Trees are BSTs where each node has an additional color attribute which can be either Red or Black. By constraining node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other. Red-black trees are approximately balanced (h = O(lg n))

1. Every node is either red or black
2. The root is black
3. 3. Every leaf (Nil) is black
4. 4. If a node is red, then both its children are black
5. 5. For each node, all simple paths from the node to descendant leaves contains the same number of back nodes.

Another important different is that instead of the leaves just being nil pointers, they all point to a nil node. This node is called T.nil.



To preserve these properties extra operations are needed when inserting or deleting elements.

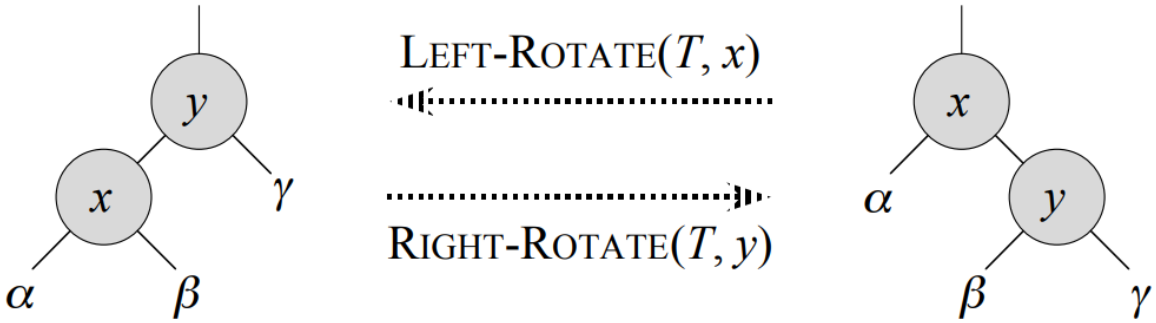
For this we use LEFT-ROTATE and RIGHT-ROTATE.

LEFT-ROTATE can be seen here, where RIGHT-ROTATE is the same, just with left and right swapped.

These are both subroutines of RB-INSERT

What this operation does, can be seen below.

```
LEFT-ROTATE(T, x)
1   y = x.right                      ← set y
2   x.right = y.left
3   if y.left ≠ T.nil                ← Turn y's left subtree (i.e., β)
4       y.left.p = x                   into x's right subtree
5   y.p = x.p
6   if x.p == T.nil
7       T.root = y                   ← Link x's parent to y
8   elseif x == x.p.left
9       x.p.left = y
10  else x.p.right = y
11  y.left = x                       ← Put x on y's left
12  x.p = y
```



Inserting into Red-Black Trees

The lines 1-13 are identical to TREE-INSERT and will not be explained here.

After insertion z.left and z.right is set to point to our nil leaf node T.nil, and the new element is colored red.

At this point we have inserted the element, but we still need to restore the RBT properties. This is where RB-INSERT-FIXUP comes in.

Runtime:

The run time is Θ(h) as 3-7 at most takes Θ(h)

And RB-INSERT-FIXUP takes Θ(h)

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == T.nil
10        T.root = z
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

RB-INSERT-FIXUP

Since only 2 of the properties can be broken of this, only these are being fixed by these procedures. The properties that can be broken by this are:

- The root is black
- If a node is red, then both its children are black

> **Basic Idea**
>
> We start from the inserted element z (which is initially at the bottom of the tree) and we move up the violations of the RB property up to the root

The function runs can be seen here.

The while loop moves the violation up in the tree preserving the invariant

There are two symmetric procedures respectively if z's parent is a left child or a right child

Each are subdivided in 3 cases where Case 2 redirects to Case 3

Examples of each case can be seen in lecture 8

Runtime:

RB-INSERT-FIXUP takes Θ(h) since the number of loops is bound by h

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    Case 1
 6              y.color = BLACK
 7              z.p.p.color = RED
 8              z = z.p.p
 9          else if z == z.p.right
10                  z = z.p                          Case 2
11              LEFT-ROTATE(T, z)
12              z.p.color = BLACK                    Case 3
13              z.p.p.color = RED
14              RIGHT-ROTATE(T, z.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

Deleting from Red-Black Trees

This function is quite like TREE-DELETE with some additional code which keeps track of the nodes x and y

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

If z's left (or right) child is Nil

- $y = z$ is the node removed, and
- x is the node that takes y's old position in T

If z has both children Nil

- y is the node that takes z's position, and
- x is the node that takes y's old position in T

- Here y has been removed or has taken z's place and colour
- If y's original colour was black then some RB property may be violated

When this function has run, the RB property must be restored. This is done RB-DELETE-FIXUP

RB-DELETE-FIXUP is only needed if y's original color was black. If it was read, no RB properties were violated

RB-DELETE-FIXUP

To understand how RB-Delete-Fixup works we shall first determine what violations of the RB property are introduced. The potentially violated properties are.

- The root is black
- If a node is red, then both its children are black
- For each node, all simple paths from the node to descendant leaves contains the same number of back nodes.

We can fictionally fix the third one by counting x as one or more black tokens. Hence, can be 'doubly black' or 'red-and-black'.

The while loop moves the extra black token (pointed by x) up in the tree

There are two symmetric procedures respectively if x is a left child or a right child

Each are subdivided in 4 cases

Basic Idea

We start from x and we **move up the extra black token** up the tree until:

- x points to a red-and-black node, in which case we colour x just back
- x points to the root, in which case we colour x just black; or
- Having performed suitable rotations and recolouring the exit the loop

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK
 6              x.p.color = RED
 7              LEFT-ROTATE(T, x.p)                    Case 1
 8              w = x.p.right
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED
11              x = x.p                                Case 2
12          else if w.right.color == BLACK
13              w.left.color = BLACK
14              w.color = RED
15              RIGHT-ROTATE(T, w)                     Case 3
16              w = x.p.right
17          w.color = x.p.color
18          x.p.color = BLACK
19          w.right.color = BLACK                      Case 4
20          LEFT-ROTATE(T, x.p)
21          x = T.root
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

Each case is visualized in slides from lecture 8

Algorithm Correctness

Loop Invariants

We use loop invariants to help us understand why an algorithm is correct.

We must prove three things about a loop invariant for an algorithm to be correct

1. **Initialization**: it holds true prior to the first iteration of the loop
2. **Maintenance**: if it is true before an iteration of the loop, then it remains true before the next iteration
3. **Termination**: when the loop terminates, the invariant gives a useful property that helps show that the algorithm is correct

Insertion sort.

The loop invariant in insertion sort is:

"At the start of each iteration of the for-loop, the sub-array A [1...j – 1] consists of the element originally in, but in sorted order A [1...j – 1]."

Initialization:

At initialization j =2 so only the elements in A [1...1] needs to be sorted. This is only one element, so it is trivially sorted.

Maintenance:

The algorithm works by moving elements one to the right, starting from the element in position j-1. this continues until it finds a value smaller than the element originally stored in position j. then that element is inserted. Since the subarray is sorted before this operation, this is the correct position for the element, so the subarray is still sorted. Then when j is incremented and we are ready for the next iteration, the loop invariant is preserved.

Termination:

The for loop terminates when j>A.length. this happens when j = n+1.

When we substitute this j into our loop invariant, we get that "the sub-array A [1...n] consists of the element originally in, but in sorted order A [1...n]" this is the entire array which is now in sorted order. With this we conclude that the sorting algorithm is correct.

Merge Sort

The loop invariant of merge sort is:

This is regarding to for loop in MERGE lines 10 to 17

"At the start of each iteration of the for loop, the subarray A [p... k – 1] contains k-p the smallest elements of L [1...n1 + 1] and R [1...n2 + 1] in sorted order. Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A."

Initialization:

we must show that the loop invariant holds true before initialization.

Before the first iteration k=p and i=j=1 therefore we can see that A [p... k – 1] is an empty array, which is implicitly sorted. Since both i and j is 1 both L[i] and R[j] smallest elements of their arrays not been copied back into A

Maintenance:

we must show that each iteration keeps the loop invariant.

There are two different cases that can happen in the loop. L[i]<= R[j] or L[i]> R[j]

If L[i]<= R[j] then L[i] is the smallest element not yet copied into A. we know A [p... k – 1] contains k-p smallest elements, so when copying L[i] into A we have the k-p+1 smallest elements in A [p... k] when we increment k and i we restore the original loop invariant.

if L[i]> R[j] we can use a similar argument, but we replace L[i] with R[j]

Termination:

the loop terminates at k=r+1

By the loop invariant, the subarray is A [p…k-1] = A [p…r] contains the k − p = r − p + 1 smallest elements of L and R in sorted order. L and R together includes r − p + 3 elements (remember the two infinity tokens) when we have now moved the r − p + 1 smallest elements from L and R to A and the largest elements are our infinity tokens, we have inserted all the elements from L and R into A in a sorted order. (except infinity tokens)

To prove the correctness of MERGE-SORT one more step is needed using the inductive method. This can be seen in Merge Sort **in Inductive method**

Inductive method

Merge Sort

To prove the correctness of Merge-Sort we prove that the subarray is sorted after a call to MERGE (this is done in Merge Sort in loop invariant)

We proceed by induction on n = r – p

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Base Case: When n=0 (r=p) the array is trivially sorted

Inductive Step:

When n>0 that means p < r and q-p<n and r-q+1<n (we can split the array into two smaller subarrays) By inductive hypothesis after calling Merge(A, p ,q) and Merge(A, q+1, r) these two subarrays are sorted. This means the required conditions for MERGE is met and the two adjacent subarrays can be merged such that A [p…r] is sorted

Runtime Analysis

To analyze the runtime of an algorithm each operation must be assigned a cost. This cost is then multiplied by the amount of times this operation has been done.

Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

We assign the costs as c values since they are dependent on which system they are run upon.

Note that when a for or while loop exits, the test is executed one time more than the loop body.

The times in the second while loop is because the while loop is accessed n-1 times and each time goes from j to the elements correct position.

This nets us a runtime defined from the following equation:

$$T(n) = c_1 n + (c_2 + c_4 + c_8) * (n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

Best Case analysis

In the best-case analysis, the first thing to do is figure out under what conditions the algorithm terminates the fastest, in the case of insertion that is when t_j is minimal for each j = 2, 3…, n. That is when the array is already sorted and $t_j = 1$ for all j = 2, 3…, n.

In this case the above formula can be reduced to this:

$$T(n) = c_1 n + (c_2 + c_4 + c_5 + c_8) * (n-1)$$

Worst Case Analysis

In the worst-case analysis, it is needed to determine under what condition the algorithm takes the longer, in the case of insertion sort that is when t_j is maximal for each j = 2, 3…, n. That is when the array is in reverse sorted order (decreasing order)

In this case we must compare A[j] with the entire sub-array A [1...j − 1]. Thus having $t_j = j$ for all j = 2, 3…, n

In this case the equation looks like this:

$$T(n) = c_1 n + (c_2 + c_4 + c_8) * (n-1) + c_5 \sum_{j=2}^{n} j + (c_6 + c_7) \sum_{j=2}^{n} (j-1)$$

We can reduce this using the rule that states $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Our operation starts from two though. For this to work we need to subtract the value for j=1, which is 1 since the array of 1 is trivially sorted.

The same can be done with the second element. But first it must be split into two functions.

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

The resulting function is

$$T(n) = c_1 n + (c_2 + c_4 + c_8) * (n-1) + c_5 (\frac{n(n+1)}{2} - 1) + (c_6 + c_7)(\frac{n(n-1)}{2})$$

This is the same as.

$$T(n) = \left(\frac{c_5 + c_6 + c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + c_8 + \frac{c_5 + c_6 + c_7}{2}\right) n - (c_2 + c_4 + c_5 + c_8)$$

When looking at this expression we can easily see it is a quadratic function following $an^2 + bn + c$ where a b and c is constants calculated from the c values.

Merge Sort

Merge sort is split into two elements, where one is called MERGE and the other is MERGE-SORT. We first analyze MERGE

A meaningful size of an instance is $n = r - p + 1$, i.e., the number elements to merge

```
MERGE(A, p, q, r)
1  n1 = q − p + 1                                          Θ(1)
2  n2 = r − q
3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4  for i = 1 to n1                                         Θ(n1 + n2) = Θ(n)
5      L[i] = A[p + i − 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  L[n1 + 1] = ∞                                           Θ(1)
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r                                          Θ(r − p) = Θ(n)
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

We can clearly see that some of the elements take constant time and a few take Theta(n)

As such we can see this algorithm takes Theta(n) time, which says its linearly dependent on n

MERGE-SORT is more complicated though since it uses recurrence

To simplify the solution, we assume that N is a power of two. This allows us to simplify calculations for the halving of the subproblem.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- **Divide:** computing the middle of the subarray takes $\Theta(1)$
- **Conquer:** solving recursively two subproblems each of size $n/2$, contributes $2T(n/2)$ to the running time
- **Combine:** the merge takes $\Theta(n)$ on an n-elements subarray.

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

We rewrite this to represent the expression without theta in the recurring part. C represents the constant that is hidden using the theta notation.

$$T(n) = \begin{cases} \theta(1) & if\ n = 1 \\ 2\,T\left(\frac{n}{2}\right) + cn \end{cases}$$

The function splits the function into smaller and smaller subproblems until we hit our base case.

This is visualized in Figure 2Figure 3

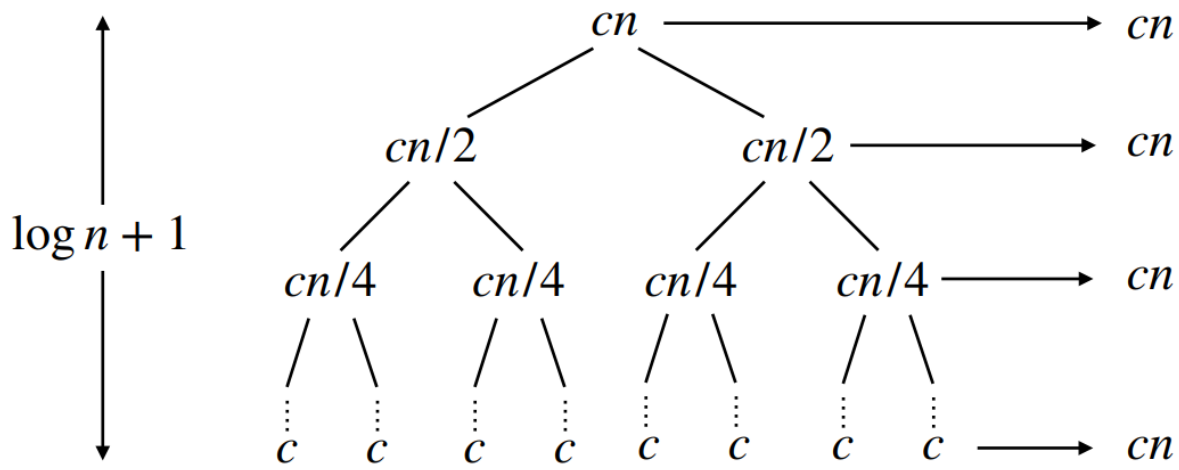


FIGURE 2

The amount of subproblems we solve is given by the height of this tree, which is log n + 1. This can also be found mathematically by using the master theorem.

Order of growth Asymptotic analysis

Often, it is not worth determining the exact running time

For large enough inputs, the multiplicative constants (coefficients) and the lower order terms are dominated by the effect of the input size itself. This allows us to abstract away the rest. And thus, we can study the asymptotic efficiency

GOAL: to simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware. This is constants like the cs from the exact runtime analysis.

This is supposed to capture the essence of the running time T(n), this is defined in the domain of natural numbers, which means it is described only using integers.

The notation is supposed to simplify comparing algorithms and accessing which algorithm fits, and which is better for a given task.

Big Theta Notation

As an example, insertion sorts run time was given by the following formula.

$$T(n) = \left(\frac{c_5 + c_6 + c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + c_8 + \frac{c_5 + c_6 + c_7}{2}\right)n - c_2 + c_4 + c_5 + c_8$$

This can be simplified to just the quadratic component and would be written as $\theta(n^2)$

This is called theta notation.

Definition: For a function $g(n)$, we denote by $\Theta(g(n))$ the set of functions

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}.$$

The value can be found by only evaluating the highest order term like $n^2 + 7899 * \mathrm{n} = \theta(n^2)$

The higher order term can be found using Figure 3

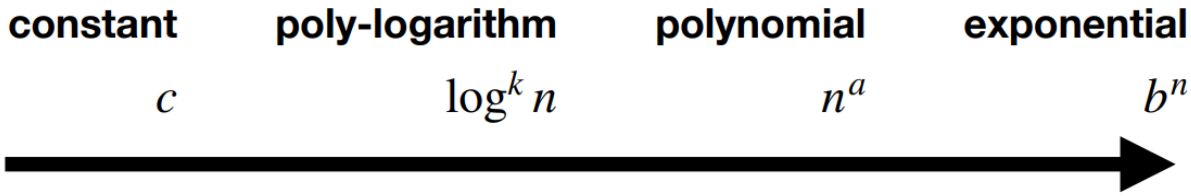


FIGURE 3

Big O-Notation

Big o is defined as:

Definition: For a function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

- $f(n) = O(g(n))$ means $f(n) \in O(g(n))$
- Asymptotically **upper bound**
- You may read as "$f(n)$ grows asymptotically slower than $g(n)$"

In practice this means that no matter what value n takes the big o notation for that number is always bigger or equal. $n^2 + n = O(n^3)$

Big Omega Notation

Big omega is defined as

Definition: For a function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}.$$

- $f(n) = O(g(n))$ means $f(n) \in O(g(n))$
- Asymptotically **lower bound**
- You may read as "$f(n)$ grows asymptotically faster than $g(n)$"

In practice this means that no matter what value n takes the big omega for that n is always lower or equal. $n^2 + n = \Omega(n^2)$

Asymptotic comparison

Theorem: For any two functions $f(n)$ and $g(n)$, we have that
$$f(n) = \Theta(g(n)) \text{ iff } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

This also tells us that big Theta is the best approximation since it is both asymptotically bigger and smaller.

This has some rules that can be useful at times.

Transitivity:

$$\begin{aligned} f(n) &= \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)) \\ f(n) &= O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)) \\ f(n) &= \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)) \end{aligned}$$

Reflexivity:

$$\begin{aligned} f(n) &= \Theta(f(n)) \\ f(n) &= O(f(n)) \\ f(n) &= \Omega(f(n)) \end{aligned}$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))$$

Transposed Symmetry:

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))$$

Remark:

When using asymptotic notation, we do not follow the usual rules for algebra.

That means we can still remove all the elements other than the one with the highest order. We also remove ALL constants. $\theta(n^2) + \theta(n^2) + \theta(n) + \theta(n) = 2\theta(n^2) + 2\theta(n) = \theta(n^2)$

Solving Recurrences

The order of growth of recurring algorithms often follow this form:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c\ , \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\ . \end{cases}$$

A is amount of subproblems created per recurrence D(n) is the time it takes to divide into said subproblems and C(n) is the time it takes to solve each subproblem. C is the size where the problem gets trivially solved (in sorting this is often when we only have one element)

There is no straightforward solution for recurrences in general, but there are 3 methods which can help

1. **substitution method**: we guess a bound and use mathematical induction to prove our guess correct
2. **recursion-tree method** converts the recurrence into a tree whose nodes represent the costs at various levels of the recursion. Usually, one uses techniques for bounding summations to solve the recurrence
3. **master method** provides bounds for recurrences of the form T(n) = aT(n/b) + f(n) where a ≥ 1, b > 1.

Substitution method

This method has two steps.

1. Guess the form of the solution
2. Use mathematical induction to find the constants and show that the solution works

A good guess is required for this method to work

we will now show how this works through an example solution on the following recurrence:

$$T(n) = \begin{cases} 1 & if\ n \leq 1 \\ 2\ T\left(\frac{n}{2}\right) + n & if\ n > 1 \end{cases}$$

First, we need to establish an **upper bound** on the recurrence

We will start off with the guess T(n) = O (n log n)

We then need to find a c>0 and a $n_0 > 0$ such that T(n) ≤ cn log n for **all** n ≥ n0

This gives us a format for an inductive hypothesis:

Inductive step:

$\mathrm{T(m)} \leq \mathrm{cm\ log(m)}$ holds for all $n_0 \leq \mathrm{m} < \mathrm{n}$ **this is the inductive hypothesis.**

We must prove $2\mathrm{T}(\lfloor \mathrm{n}/2 \rfloor) + \mathrm{n} \leq \mathrm{cn\ log(n)}$

To do this we set $m = \lfloor \mathrm{n}/2 \rfloor$ which gives us

$$\mathrm{T}(\lfloor \mathrm{n}/2 \rfloor) \leq \mathrm{c}\lfloor \mathrm{n}/2 \rfloor \log(\lfloor \mathrm{n}/2 \rfloor)$$

We substitute this into the original recurrence.

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + n && (\text{def } T) \\
&\leq 2c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor) + n && (\lfloor n/2 \rfloor < n \text{ and Ind.Hp.}) \\
&\leq cn \log(n/2) + n && (c > 0 \text{ and } \lfloor n/2 \rfloor \leq n/2) \\
&= cn \log n - cn \log 2 + n \\
&= cn \log n - cn + n \\
&\leq cn \log n && (\text{assuming } c \geq 1)
\end{aligned}
$$

This proves the induction step for the upper bound. cn log n is always bigger than T(n)

It is important to note that we need to prove the EXACT form of the inductive hypothesis. And not just that they have the same asymptotic notation

Base Case

Typically, we need to show that the hypothesis holds for the base case of the recurrence

But we have a problem T (1) = 1 but c log 1 = 0.

But since this is an asymptotic function, we can start from any value for n_0 we see that from $n > 3$ the recurrence no longer depends directly on t(1) which means we can solve this issue by setting $n_0 = 2$ and set our base cases as T(2)=4 and T(3)=5 **Both these cases are needed to avoid running into T(1) . It is a good fit since they can both easily be derived from T (1) as well**

We then find a C value so these base cases both fulfill the following:

$$\mathrm{T}(2) \leq \mathrm{c}2 \log 2 \quad \text{and} \quad \mathrm{T}(3) \leq \mathrm{c}3 \log 3$$

It can from this be calculated that c>=2 works

Substitution method tips

Sometimes a little algebraic manipulation can make an unknown recurrence like one you have seen before

Example:

Consider the recurrence $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$

- Define $m = \log n$ (let's not worry about rounding off values)
- Changing variable yields to $T(2^m) = 2T(2^{m/2}) + m$
- We can further rename $S(m) = T(2^m)$
- Producing $S(m) = 2S(m/2) + m$ and we know $S(m) = O(m \log m)$
- Substituting back we obtain $T(n) = O(\log n \log \log n)$

this can vastly simplify an expression into much more manageable bits.

Making a good guess

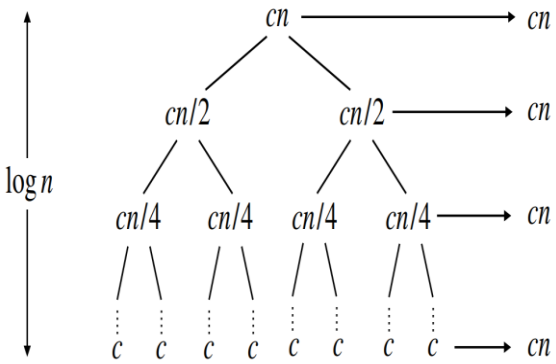
There is no simple way to always make a good guess but using recursion trees can help if it does not look like something well known. **If it looks like a solution you know the answer to, try using that solution as your guess**.

Recursion Tree model

The recursion tree model converts the recurrence into a tree whose nodes represent the costs at various levels of the recursion. Usually, one uses techniques for bounding summations to solve the recurrence.

Constructing the tree consists of 3 things

1. Each node represents the cost of a single subproblem in the unravelling of recursive function invocations
2. We sum the costs within each level obtaining peer-level costs
3. We sum all the peer-level costs obtaining the total cost



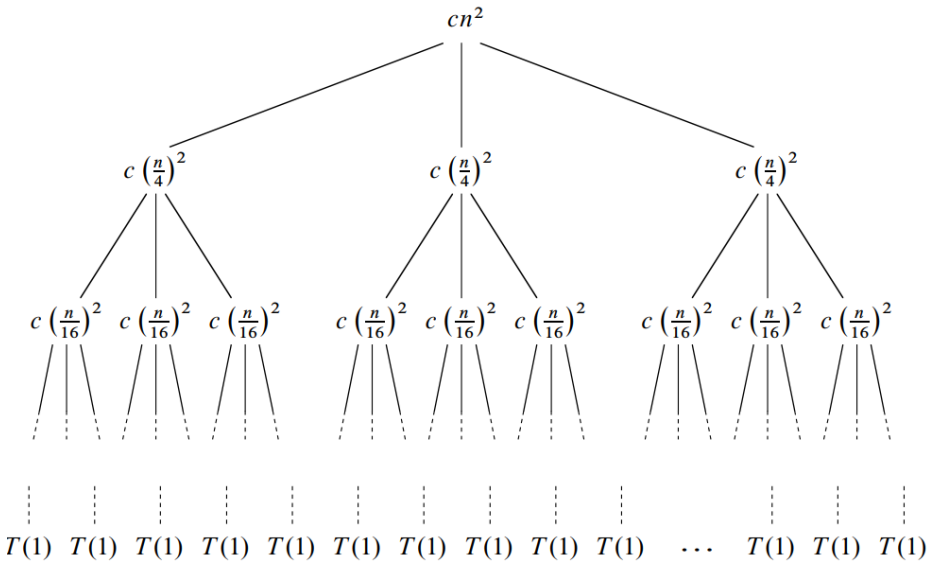
Recursion tree model explained through example:

We want to provide a good guess for the recurrence

We focus at finding an upper bound for the solution.

We assume that n is a power of 4. This simplifies the solution because we each time split into subproblems a quarter of the original size.

We create a recursion-tree for while keeping in mind c>0.



We then calculate the height of the tree.

The recurrence cut the problem by a factor of 4 each recursive step, this means the subproblem size of a node at depth i is n/(4)^I so when we hit the base case the size of the problem is 1 which means 1 = n/(4)^i we then isolate i which gives us **i=log4(n)**

Since this is the depth of the tree, we can see the height is log4(n) +1

Next, we find the peer level cost (the cost of all subproblems of the same size, these are at the same depth in the tree)

To do this we need to find the general cost of the i-th peer level

What is the i-th peer-level cost?

- Each level has 3 times more nodes that the previous, thus there are 3^i nodes at level i
- The subproblem size of a node at depth i is $n/4^i$ thus has a cost $c(n/4^i)^2$
- Thus the peer-level cost is $3^i c(n/4^i)^2 = (3/16)^i cn^2$

This is in practice just the cost of each node multiplied with the number of nodes.

We then calculate the peer level cost of the bottom level.

What is the cost at bottom-level?

- The bottom level is $i = \log_4 n$
- It has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each of cost $T(1)$
- Since $T(1) = \Theta(1)$, the total cost of the bottom level is $\Theta(n^{\log_4 3})$

We then sum up the costs of all the peer levels, which will lead us to the upper bound.

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \quad < \sum^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \quad = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \quad = O(n^2)$$

This can then be used as a guess for our guess in the substitution method.

Master Method

The master method gives us an easy way to solve recurrences of a certain form:

T(n) = aT(n/b) + f(n)

Where a>=1 b>1 and f(n) is an asymptotically positive function.

It is much simpler than the substitution method and is as such preferred if possible.

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

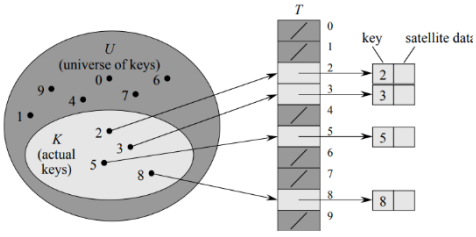
where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Direct-Address Tables

A Direct-Address table is not really a hash table, but a simplified version of one, which is a better solution in some cases where m is small. Which large m this wastes a lot of space.

Assumptions
1. Suppose that an application needs a dynamic set in which each element has a key drawn from the **universe** $U = \{0,1,2,\ldots,m-1\}$, where m is not too large.
2. We shall assume that no two elements have the same key



We use and array T[0...m-1] called a Direct-address table where each spot corresponds to a key in the universe.

The table has thee functions needed to operate.

DIRECT-ADRESS-SEARCH, ***DIRECT-ADRESS-INSERT*** and ***DIRECT-ADRESS-DELETE***

This is used several places, ex. In counting sort where the C array is a direct-address table.

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

Hash Tables

Hash Tables is a method to, under reasonable assumptions, allow searching to happen in constant time.

Hash Tables are only a possibility when the list only needs the dictionary operations ***Insert***, ***Search***, and ***Delete.*** It is also assumed that each element in the dictionary has a *key* attribute along with other possible satellite data.

A hash table is an effective data structure for implementing these dictionaries.

Direct-address tables are not a good solution when the universe of keys is large.

Storing a table T of size |U| may be impractical.

The set K of keys stored may be so small relative to U that most space allocated for T would just be wasted.

When K is much smaller than U, we can reduce the storage requirement to Θ(|K|) while maintaining the benefit that searching cost (on average) only O(1) time

This can all be done by mapping a key K to a lot using a hash function

This hash function decides where to store a key K in our table and can be used to find it again.

One problem with using hash functions is that two values might end up with the same hash value in which case there is a collision.

Ideally you would want to avoid collisions all together, but this cannot be completely avoided.

Collisions can be resolved in several different ways.

Collision resolution by Chaining

Collision resolution by Chaining places all elements with the same hash value in a linked list**.** This solves the collision, but complicates searching for an element, which can now take longer because we must search through a linked list as well.

One advantage of chaining for resolution is that the dictionary operations are easily implemented.

Insertion takes Θ(1) time if we assume that the element is not in the table. Otherwise, one should first check this (at additional cost) by searching for an element with key .key

```
CHAINED-HASH-INSERT(T, x)
1  insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
1  search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
1  delete x from the list T[h(x.key)]
```

Searching We simply use the linear search procedure for linked lists as seen in Linked List Search

Deletion We can delete the element in Θ(1) time if we use doubly linked lists. With singly linked lists deletion can take linear time according to the length of the list.

Runtime:

Worst case:

all keys hash to the same slot creating a list of length n.

int this case the run time is Θ(n) for searching the linked list and Θ(1) for computing the hash value. So, the worst-case runtime is **Θ(n)** From this we can see we don't user hash tables for their worst case, since this is strictly worse than just using a linked list.

Average case:

The average-case run-time depends on how well the hash function h distributes the set of keys among the slots m, on average.

To calculate the average case, we assume that any given key is equally likely to hash into any of the m slots. We call this assumption simple uniform hashing. ($0 \le j \le m, E[n_j] = \alpha,$ where is the length of the list) α is the load factor.

Definition: we define the **load factor** α of T as n/m, that is the average number of elements stored in a chain.

If we assume simple uniform hashing, we can assume there is the same amount of values hashed to each place in our array. This means the number of elements in each position is equal to the load factor (elements to store/positions in hash table) from this we have these two theorems.

Theorem 11.1
In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Theorem 11.2
In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

So, what does this mean?

If the number of slots in T is at least propositional to the number of elements in T (we have around 1 element in each slots) we can say that n=O(m) this gives us α = n/m = O(m)/m = **O(1)**

Overall performance Hash Tables (with chaining):

- Searching for a key: worst-case $\Theta(n)$, average-case $O(1)$
- Inserting an element: worst-case $\Theta(1)$
- Deleting an element: worst-case (doubly linked lists) $\Theta(1)$

Hash Functions

A good hash function satisfies (approximately) the simple uniform hashing assumption. Unfortunately, we have no general way to check this condition, since we rarely know the probability distribution from which the key is drawn

In practice we can often employ heuristic techniques to create hash functions that works well

Qualitative information about the distribution of keys may be useful in this design process E.g., symbol table use alpha-numeric keys, and similar names occurs often in the same program.

A good approach derives the hash value in a way that it is independent of any patterns that may exists in the set of keys.

A hash function does the following **h: U → {0,…, m − 1}** This means that a function takes a value from the universe of keys, and gives it a numeric value in the range of our array T. most **hash functions assume that the universe is the natural numbers.** If the numbers are not numbers, we must find a way to turn them into numbers.

Division method

We map a key K into one of m spots. The hash function is **h(k) = k mod m** It often works well when is a prime not too close to an exact power of 2

Example:

- Suppose we want to allocate a hash table to hold roughly $n = 2000$ character strings, where a character has 8 bits.
- We don't mind examining on average 3 elements in an unsuccessful search, an so we use $m = 701$ because 701 is a prime near $2000/3 \cong 666.67$ but it is not near any power of 2, indeed $2^9 = 512 < 701 < 1024 = 2^{10}$

Multiplication method

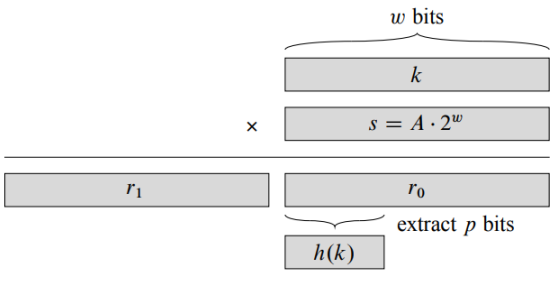
In the multiplication method for creating hash functions, we map a key into one of slots using a constant 0 < A < 1

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

Advantage: the value of m is not critical. Typically, one chooses it to be a power of 2

Example:

- Suppose the word size of the machine is w bits and that the key k fits into a single word.
- Using $m = 2^p$ for some $p \in \mathbb{N}$, and
- $A = s/2^w$ for some $s \in \mathbb{N} \cap (0,2^w)$
- Then $h(k)$ can be efficiently computed as



30

Open Addressing

In open addressing all elements occupy the hash table itself, that is each table slot contains either an element of the dynamic set or Nil. Unlike chaining, no list and no elements are stored outside the table. This has some interesting properties.

- The hash table can "fill up" so that no further insertions can be made
- The load factor can never exceed 1 ($m \geq n$)
- More slots in the same amount of memory.

To perform insertion using open addressing we successively examine, or probe, the hash table until we find an empty slot in which to put the key. Instead of following the fixed order(0,1,2…) , the order depends on the key

We extend the hash function to include the probe number as a second input:

$$h: U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$$

The probe sequence must be a permutation all numbers between 0 and m-1 so that every slot is eventually considered as candidate spot for k.

This has two methods for standard dictionary action. Insert, Search.

```
HASH-INSERT(T, k)
1  i = 0                          <- Initialise probe number
2  repeat
3      j = h(k, i)                <- Current slot
4      if T[j] == NIL             <- If the slot is free insert the
5          T[j] = k                  key, otherwise move to
6          return j                  the next probe number
7      else i = i + 1
8  until i == m                   <- We have probed the whole table
9  error "hash table overflow"       and no spot is available.
```

```
HASH-SEARCH(T, k)
1  i = 0                          <- Initialise probe number
2  repeat
3      j = h(k, i)                <- Current slot
4      if T[j] == k               <- If the slot contains the key
5          return j                  return, otherwise move to the
6      i = i + 1                     next probe number
7  until T[j] == NIL or i == m
8  return NIL                     <- We have probed the whole table
                                     and k was not found.
```

If deletion is necessary, you are better of using chaining instead. Explanation can be found in lecture 7 slides 38-40

Runtime:

We express our analysis in terms of load factor $\alpha = n/m$

Assumption: we use uniform hashing which means the probe sequence used to insert or search of each key k is equally likely to be any permutation of the numbers between 0 and m-1

The analysis studies the expected number to probes for hashing with open addressing under uniform hashing.

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

The analysis studies the expected number to probes for hashing with open addressing under uniform hashing.

Intuitive Interpretation:

$$1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$$

With probability α, the first probe doesn't find a free slot, so we need to probe a second time

E.g. if the table is $50\,\%$ full, the average number of probes is at most

$$1/\alpha \ln 1/(1-\alpha) = 2 \ln 2 \cong 1.387$$

If the table is $90\,\%$ full, we have

$$1/\alpha \ln 1/(1-\alpha) = 10/9 \ln 10 \cong 2.559$$

True uniform hashing is very difficult to implement

We present three commonly used techniques:

Linear Probing

Linear probing uses an auxiliary hash function h': U → {0…, m − 1}

This is used to in the actual hash function which is $\mathrm{h(k, i)} = (\mathrm{h'(k)} + \mathrm{i}) \bmod \mathrm{m}$

This takes the result from the auxiliary function, checks if its free if not, we try a higher i value. This moves linearly through the entire array.

The initial probe determines the entire sequence, hence we have only m distinct probe sequences

It suffers from a problem known as primary clustering: long runs of occupied slots build up, increasing the average search time.

Quadratic Probing

Quadratic probing uses an auxiliary hash function h': U → {0, … , m − 1}

This is used to in the actual hash function which is $\mathrm{h(k, i)} = (\mathrm{h'(k)} + c_1 i^2 + c_2 i^2) \bmod \mathrm{m}$

where c1 and c2 are constants

The initial probe determines the entire sequence; hence we have only m distinct probe sequences like in linear probing. Better than linear probing, but c1 ,c2 and m must be chosen wisely

Suffers from milder form of clustering, called secondary clustering.

Double Hashing

Double hashing uses two auxiliary hash functions.

The hash function is $\mathrm{h}(\mathrm{k},\mathrm{i}) = (h_1(\mathrm{k}) + \mathrm{i}\, h_2(\mathrm{k})) \bmod \mathrm{m}$

Unlike the previous methods, the probe sequence is not entirely determined by the first probe

For suitable h_1 and h_2, the number of possible probe sequences is Θ(m2) since each pair$(h_1(k), h_2(k))$ can yield a distinct one.

The performance gets very close to the "ideal" scheme of uniform hashing

Dynamic Programming

Dynamic programming, like divide-and-conquer, solves problems by combining the solutions of subproblems.

It applies when the subproblems overlap (i.e., when subproblems share subproblems)

Typically applied to optimization problems:

- Problems that admit many possible solutions
- Each solution has a value (or cost)
- One wants to find an optimal solution with max value (or min cost)

When developing a dynamic-programming algorithm we follow a sequence of four steps:

1. **Characterize the structure of an optimal solution**
2. **Recursively define the value of an optimal solution**
3. **Compute the value of an optimal solution, typically in a bottom-up fashion**
4. **Construct an optimal solution from computed information**

Basic Idea

- We arrange for each subproblem to be solved only once, storing its solution
- If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it

Dynamic programming uses additional memory to save computation time

There are two approaches to dynamic programming:

1. Top-down with memorization
2. Bottom-up

The following methods will be applied to a rod cutting problem to visualize the solution.

Rod-Cutting

The rod cutting problem is an example of a use of dynamic programming.

> **The rod-cutting problem:** given a rod of length n inches and a table of prices p_i for $i = 1,2,\ldots,n$, determine the maximum revenue r_n obtainable by cutting the rod an selling the pieces.

Since this, with a large table, and a long rod, can have an insurmountable amount of solutions, it would take too much computing power to solve this with brute force.

Top-Down with Memorization

The top down method is written recursively in a manner close to the brute force variant. **The difference is that when a subproblem has been calculated, it is stored in an array or hash table for future lookup**. The procedure now first checks to see whether it has previously solved the subproblem, if it has, it returns the saved value, otherwise it calculates it as usual, and saves the value in the array.

We say that the recursive procedure has memorized what results it has computed previously

An example can be seen here

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

The array stores the solutions of the subproblems. $-\infty$ stands for "unknown result"

Call auxiliary procedure

First an array is filled with token values. These means the values have not been calculated yet. Then we call the function below with this array along with the original parameters.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

Recover stored optimal value

Same as ROD-CUT

The first thing that is checked, is if the solution has been calculated, if it has, we return it. (line 1-2)

Else we check if the length of the rod to solve for is 0, if it is, we set q to 0, store the value, and return it.

If we have not computed the solution before it is done as in the original ROD-CUT function. The computed solution is then stored and returned.

Runtime:

1-3 of MEMOIZED-CUT-ROD is a for loop which runs n times. Θ(n)

MEMOIZED-CUT-ROD-AUX is called n times recursively, and each time a for loop which runs n times is used. **This gives Θ(n2) runtime**

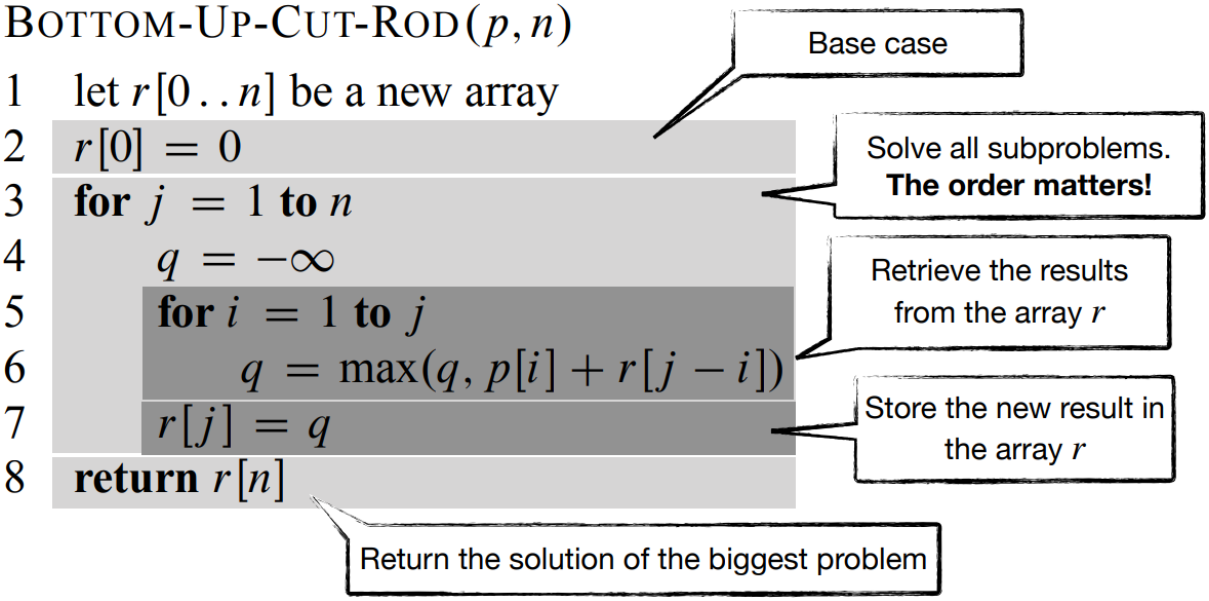
Bottom-up method

Depends on some natural notion of the "size" of the subproblem: solving any particular subproblem depends only on solving "smaller" subproblems

We sort the subproblems by size and solve them in size order, smallest first.

When solving a subproblem, we have already solved all the smaller subproblems its solution depends on and we have saved their solutions

An example can be seen here



The function makes a new arrau, and sets the value corresponfing to a rod length of 0 to 0 (we cannot sell any rod if we don't have any).

We then go through every length of rod lower than n, starting from 1. When this has been calculated, we store the value in the array. When j=n and the for loop breaks, the optimal solution is tores in r[n] which is returned.

Runtime:

The runtime is given from the two nested for loops. Their run time is defined by $\sum_{j=1}^{n} \Theta(j) = \Theta(n^2)$ so **the runtime is $\Theta(n^2)$**

Graph Theory

A graph is a pair G = (V, E) where

- V is a set of vertices
- $E \subseteq \{\{u, v\}: u, v \in V\}$is a set of two-sets of vertices, whose elements are called edges

The vertices u and v of an edge$\{u, v\} \in E$ are called endpoints of the edge.

The edge $\{u, v\} \in E$ is said to be incident on the vertices and

There are several ways to represent edges in a Graph, but most common are Adjacency-list and Adjacency-matrixes.

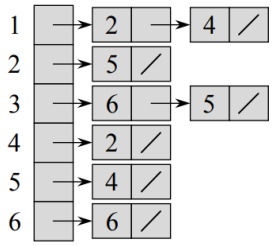
Adjacency lists

The adjacency-list representation of $G = (V, E)$ consists of an array (or an hash table) Adj of $|V|$ list

For each vertex $u \in V$, $Adj[u]$ consists of all the vertices adjacent to u, i.e., $v \in Adj[u]$ iff $(u, v) \in E$

We will treat the array Adj as an attribute of the graph G

It requires $\Theta(|V| + |E|)$ memory



Adjacency matrix

The adjacency-matrix representation of $G = (V, E)$ consists of a $|V|^2$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \textbf{if } (i, j) \in E \\ 0 & \textbf{otherwise} \end{cases}$$

It requires $\Theta(|V|^2)$ memory

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Breadth First Search (BFS)

BFS is a simple search algorithm for graphs.

It works by systematically exploring all edges from a single vertex V before moving to the next, this continues until it has found all vertices reachable from V

It computes the distance (smallest number of edges) from any reachable vertex
It also produces a “breadth-first tree” T rooted in S that contains all vertices reachable from S.

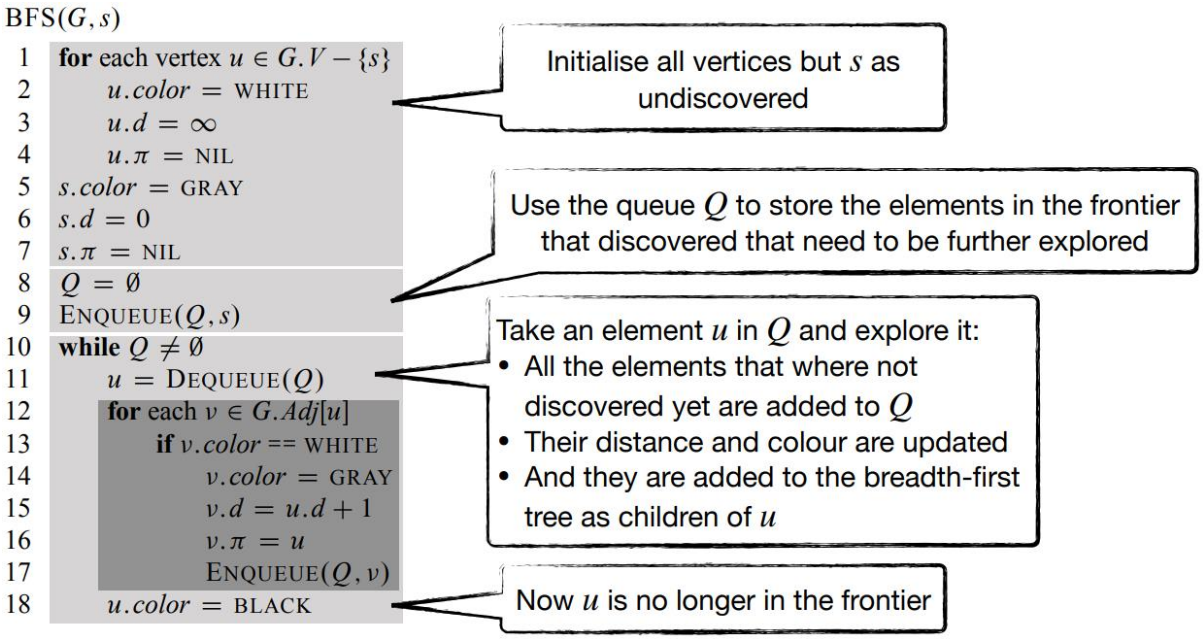
The simple path in T from s to any other v ∈ T corresponds to a shortest path from s to v in G

The algorithm works both for directed and undirected graphs

Starting from s, it expands the frontier between discovered and undiscovered vertices uniformly across the frontier.

It discovers all vertices at distance k from s before discovering any vertices at distance k+1

Wherever the search discovers a new vertex v in the adjacency-list of a vertex u in the frontier, the vertex v and the edge (u, v) are added to the breadth-first tree



Runtime:

The overhead for initialization is O(|V|) this is lines 1-4, where the runtime comes from the for loop

8-18 requires a bit more analysis. After initialization BFS never whitens a vertex, thus the test in line 13 ensures that each vertex in enqueued at most once, this means that the total time for queue operations is O(|V|).

BFS scans each adjacency-list at most once so the total length of all adjacency lists is Θ(|E|), thus total time for scanning them is Θ(|E|)

Total: O(|V| + |E|)

Theorem 22.5 (Correctness of breadth-first search)

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

Depth First Search (DFS)

Depth-first search explores edges out to the most recently discovered vertex v that still has unexplored edges leaving it.

The strategy searches "deeper" in the graph whenever possible.

When all v's edges have been explored, it "backtracks" to explore edges leaving the vertex from which v was explored

Like BFS, it records the exploration by using predecessor attributes, creating a depth-first forest $G_\pi = (V, E_\pi)$ where $E_\pi = \{(v.\pi, v): v \in V \text{ and } v.\pi \neq NIL\}$

The algorithm works both for directed and undirected graphs

DFS works by starting from a vertex, discover all its edges, and proceeding recursively on all adjacent vertices which have not been discovered yet.

It keeps track of the process by coloring vertices

- White vertices have not been discovered yet
- Grey vertices have been discovered but not totally explored
- Black vertices have been totally explored

These properties mean that a black and a white vertex can never be connected.

Additionally, DFS records when it discovers vertex by setting a timestamp:

u.d records the time when u becomes grey

u.f records the time when u becomes black

For each vertex u.d < u.f (a vertex is always discovered before it can be finished.)

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

Initialise all vertices as undiscovered and "start the timer"

Start the exploration on a vertex. Line 7 starts a new depth-first tree

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

Mark u as discovered and increase the timer

Explore adjacent vertices by recursively visit those not discovered yet

After returning from the recursive calls u is marked as totally discovered

The result my depend upon the order in which vertices are selected in DFS(G) and edges are explored in DFS-VISIT(G, u)

Runtime:

DFS: The first for loop takes Θ(|V|) and we can use aggregate analysis to see that DFS-Visit is called exactly once per vertex

DFS-VISIT the for loop goes through every edge for a vertex. But since a vertex becomes gray, we only make subsequent calls if a vertex has not already been discovered. That means we have an upper bound of $\sum_{v \in V} |\mathrm{Adj}[v]| = \Theta(|E|)$ (we at most use each edge once)

TOTAL: Θ(|V| + |E|)

DFS properties:

The structure of the depth-first forest exactly mirrors the structure of the recursive calls of DFS-Visit — $(u, v) \in E_\pi$ iff DFS-Visit(G, v) was called during a search of u's adjacency-list

Discovery and finishing times have "**parenthesis structure**"

Theorem 22.7 (Parenthesis theorem)

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.

Corollary 22.8 (Nesting of descendants' intervals)

Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $u.d < v.d < v.f < u.f$.

Theorem 22.9 (White-path theorem)

In a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

DFS Classification:

DFS can be used to classify the edges of $G = (V, E)$ relative to the depth-first search forest $G_\pi = (V, E_\pi)$

- **Tree edges**: edges in the depth-first forest G_π
- **Back edges**: $(u, v) \in E$ connecting a vertex u to an ancestor in G_π
- **Forward edges**: $(u, v) \in E \backslash E_\pi$ connecting u to an descendant in G_π
- **Cross edges**: all other edges. Connect vertices across the forest

Lemma 22.11

A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

Topological sort

DFS can be used to perform a topological sort of a direct acyclic graph (DAG)

The topological sort of a DAG is a linear ordering f: $V \rightarrow \mathbb{N}$ of all its vertices such that f is injective and for any (u, v) ∈ E f(u) is smaller than f(v) (f(u) ≤ f(v))

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Runtime:

Topological sort has the same runtime as DFS. Explanation can be seen in that chapter. **Θ(|V| + |E|)**

Correctness TODO (if not done can be found in lecture 10 slides about 75% through)

Strongly Connected Components:

Definition: Given a *directed* graph $G = (V, E)$, we say that $C \subseteq V$ is a *strongly connected component* (SCC) if it is a *maximal* set of vertices such that for every $u, v \in C$ we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$, i.e., v can be reached from u and vice versa.

Our algorithm for finding SCCs of a graph uses the transpose of the graph, which is defined as the same vertices, but all edges reversed.

Given an adjacency-list representation of G one can compute G^T in O(|V| + |E|) time

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^{T}
3 call DFS(G^{T}), but in the main loop of DFS, consider the vertices
in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
separate strongly connected component

Runtime: O(V+E) since DFS takes this time and computing the transposed does too. (assuming adjacency list and not adjacency matrix.)

Correctness TODO (if not done can be found in lecture 10 slides about 90% through)

Weighted graphs:

A weighted graph is a graph with a weight function w: $E \rightarrow \mathbb{R}$ assigning a weight with each edge. The notion extends both to directed and undirected graphs

Adjacency-list representation

- Add weight satellite information on the elements in the lists
- $(u, v) \in E$ iff $Adj[u] \in \langle w, w(u, v) \rangle$

Adjacency-matrix representation

- Boolean entries in the matrix $A = (a_{ij})$ are replaces by weights
- $a_{i,j} = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \text{Nil} & \text{otherwise} \end{cases}$
- Depending on the application it may be convenient to use 0 or ∞ in place of Nil

Shortest Path

It is often beneficial to find the shortest path between two vertices.

Given a directed graph with a weight function, the weight of a path is the sum of the weight of its constituent edges$\mathrm{w(p)} = \sum_{i=1}^{k} \mathrm{w(v}_i - 1, \mathrm{v}_i)$

Definition: We define the the shortest-path weight $\delta(u, v)$ from u to v by $\delta(u, v) = \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}$. If there is no path connecting u and v then $\delta(u, v) = \infty$.

Many shortest path algorithms rely on lemma 24.1

Lemma 24.1 (Subpaths of shortest paths are shortest paths)

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

To find the shortest path between two points in a graph, it is important that **no negative cycles are present**, since these will spiral endlessly since it is more efficient to keep cycling.

With this assumption we can assume that any solution is a simple path e.g. no cycles are present.

Like BFS, we use the π attribute to represent shortest paths form to any other vertex reachable from it

Relaxation

To find the shortest path, a technique called relaxation is used. This starts by initializing all vertexes

INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

This sets the distance to all vertices to infinity (except the vertex we begin from)

After initialization we will improve the shortest-path estimates by relaxing edges.

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```

Both the Dijkstra and the Bellman-Ford algorithm uses these principles for calculating shortest path.

After each relaxation step the following properties hold:

Triangle inequality (Lemma 24.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 24.11)
We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 24.12)
If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 24.14)
If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 24.15)
If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property (Lemma 24.17)
Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Bellman-Ford Algorithm

Bellman-Ford is an algorithm made for calculating the shortest distance from a vertex to all other vertices in the tree. **An advantage of this algorithm is that it detects negative cycles**.

Main Idea

- Since simple paths have at most $|V| - 1$ edges, we discover all shortest paths from s by performing $|V| - 1$ relaxation steps for each edge $(u, v) \in E$.
- If after this we can still decrease the shortest-path estimate then it must be because of a negative cycle

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Initialise attributes of each vertex. Time $\Theta(|V|)$

Expand the frontier of shortest paths $|V|-1$ times. Time $\Theta(|V||E|)$

Check if we can further improve some the estimate. Time $\Theta(E)$

This algorithm initializes all vertices and relaxes all edges from each vertex in the graph G (line 2-4).

It iterates over all edges in G to see if there are further improvements to be made.

Runtime: Θ(|V||E|)

Correctness TODO. (lecture 11 about 60% through.)

Single-Source shortest paths for DAGs

For DAGs we do not need to worry about negative cycles because there are none.

We can simplify the algorithm by exploiting topological ordering of the vertices.

Main Idea

- In a DAG all paths follow any topological ordering of the vertices. Thus also shortest paths from s.
- We can reuse Bellman-Ford idea reducing the number of iterations of its main for loop

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

Compute a topological ordering. Time $\Theta(|V|+|E|)$

Expand the frontier of shortest paths following topological ordering. Time $\Theta(|V|+|E|)$

First the algorithm sorts the vertices in topological order. (see topological sort), we can then relax them in this order. This saves us a second runtime to look for improvements.

Runtime: Θ(|V| + |E|)

Correctness TODO (lecture 11 75% through)

Dijkstra's Algorithm

Dijkstra's algorithm Assumes no edge weights are negative, this simplifies the algorithm vastly, since there cannot be negative cycles. It generalizes BFS over weighted graphs.

Main Idea

- Like BSF expands the frontiers in order of increasing distance from s.
- Each node is visited once and its adjacent vertices
- Use a min-priority queue organised according to the actual value of the shortest-path estimates of vertices not encountered yet

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

- S = vertices completed
- Q = vertices not yet completed

Remark. Q is implicitly populated using Insert operations

Take a vertex at minimal distance from s in Q and expand the frontier from there by relaxing outgoing edges

Remark: implicitly use a DecreaseKey operation on Q when relaxing the edge

set. The algorithm maintains the invariant that Q = V \ S at the start of each iteration of the while loop.

Runtime:

INITIALIZE-SINGLE-SOURCE takes Θ(|V|)

Q is implicitly populated using |V| Insert operations

Each vertex enters Q exactly once. Then the Decrease-Key operation (implicitly called by Relax) is called at most |E| times

Q implemented using an array

- Assumption vertices are numbered 1 to $|V|$
- Insert and Decrease-Key take $\Theta(1)$
- Extract-Min takes $\Theta(V)$
- Therefore Dijkstra's algorithm running time is $\Theta(|V|^2) + \Theta(|E|) = \Theta(|V|^2)$

If G is sufficiently sparse— in particular $|E| = o(V^2/\lg V)$ one can use

- **Min-Heap:** achieving $O((V+E)\lg V)$ which becomes $O(E \lg V)$ if all vertices are reachable from s
- **Fibonacci heap:** achieving $O(V \lg V + E)$

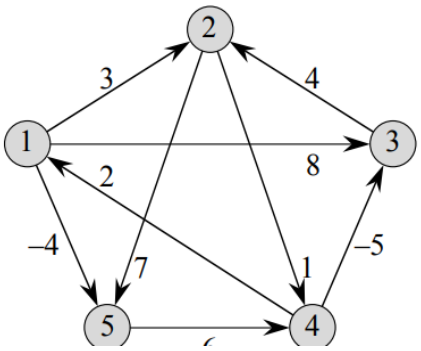
Correctness: TODO (can be found 80% through lecture 11)

All-pairs Shortest paths

All-pairs shortest path algorithms try to solve to All-pairs shortest paths problem (duh)

> **All-pairs shortest paths problem:** Given a weighted, directed graph $G = (V, E)$, with weight function $w: E \to \mathbb{R}$, we want to find for every pair of vertices $u, v \in V$ a shortest (least-weight) path from u to v.

We typically want the output to be in tabular form like the example below:



$$\begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

We can solve an all-pair shortest path problem by running a single-source shortest-paths algorithm |V| times, once for each vertex as the source.

If all edges are nonnegative, we can use Dijkstra on each vertex, which takes O(V3 + VE) = $O(V^3 + VE) = O(V^3)$.

If ANY case is negative, we must use Bellman-ford on each vertex, which takes $O(V^2E)$ which is $O(V^4)$ on dense graphs (graphs with many edges)

This can be done more efficiently using alternative methods as described below.

An important distinction between these algorithms are that the efficient ones for all pairs use adjacency-matrix representation instead of adjacency-lists

Negative edges are allowed, **but no negative cycles are allowed in the graph.**

The output is a matrix as shown above

the representation of optimal paths for all pairs of vertices is done via a predecessor matrix. The elements in this matrix is either NIL if not path between I and J is available, or the predecessor of j on some optimal path from I to j

We can print the path using the following algorithm

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
1  if i == j
2      print i
3  elseif π_ij == NIL
4      print "no path from" i "to" j "exists"
5  else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
6      print j
```

We are already at the source i

Recursive call with the predecessor of j in a shortest path from i. Then, print i

Lemma 24.1 (Subpaths of shortest paths are shortest paths)

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

This along with the principles of matrix multiplication allows a fast algorithm, but first we look at a slower, but simpler version.

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  for m = 2 to n − 1
4      let L^(m) be a new n × n matrix
5      L^(m) = EXTEND-SHORTEST-PATHS(L^(m−1), W)
6  return L^(n−1)
```

Note that $L^{(1)} = W$, so we start from here instead of $L^{(0)}$

This algorithm uses the rules of matrix multiplication to calculate the distances, it does so one step at a time, a bit like BFS does. It continues until the loop breaks at n-1. This means that we have explored the maximum amount of edges the shortest path can take (assuming no negative loops).
Assuming G has no negative cycles, the matrix $L^{(|V|-1)}$ contains the actual shortest-path weights

REMARK: this running time is Θ(|V|^4) which is NOT better than we could obtain using bellman-ford

The extend shortest paths algorithm has a lot of similarities with matrix multiplication as can be seen here

$$L' = L \odot W$$

$$l'_{ij} = \min_{1 \le k \le n} \{l_{ik} + w_{kj}\}$$

```
EXTEND-SHORTEST-PATHS(L, W)
1  n = L.rows
2  let L' = (l'_ij) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          l'_ij = ∞
6          for k = 1 to n
7              l'_ij = min(l'_ij, l_ik + w_kj)
8  return L'
```

$$C = A \cdot B$$

$$c_{i,j} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

With this realization, the algorithm can be restated as a sequence of matrix multiplications. With this can see that we can use **repeated squaring** to improve the algorithm

$$L^{(1)} = W$$
$$L^{(2)} = W \odot W = W^2$$
$$L^{(4)} = W^2 \odot W^2 = W^4$$

If we use this principle and terminate the for loop when we exceed n-1, which happens at $L^{2\lceil \lg(n-1) \rceil}$

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  m = 1
4  while m < n − 1
5      let L^(2m) be a new n × n matrix
6      L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
7      m = 2m
8  return L^(m)
```

Perform $L^{(m)} \odot L^{(m)}$ in a repeated square fashion still takes $\Theta(n^3)$

But now we iterate $\Theta(\lg n)$ times

This allows a runtime of $\Theta(n^3 \lg n)$

Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm exploits a different dynamic programming formulation to solve the all-pairs shortest-paths problem. **The algorithm runs in $\Theta(n^3)$ time.**

Negative weight edges may be present, but we assume that there are no negative cycles

The algorithm computes in a bottom-up fashion the sequence D(0) , D(1) , D(2) …, D(n)

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1))
8  return D^(n)
```

The algorithm goes through the shortest path from each vertex to each other vertex. It does this for every vertex. (k is vertices whereas I and j are going through all the current shortest paths.)

Transitive closure of a directed graph

Definition: Given a directed graph $G = (V, E)$, we define the **transitive closure** of G as the graph $G^* = (V, E^*)$ where $E^* = \{(i, j)$: there is a path from i to j in $G\}$

In other words. A transitive closure means that every vertex than can be reached from a vertex also has a direct path.

One way to compute it is to assign weight to all vertices, run the Floyd-Warshall algorithm.

Then, $(\mathrm{i}, \mathrm{j}) \in \mathrm{E}*$ if and only if $d_{ij} < \infty$

Alternatively, we can save time and space in practice (still having same asymptotic run-time) implementing a "Boolean" variant to the Floyd-Warshall algorithm.

```
TRANSITIVE-CLOSURE(G)
1  n = |G.V|
2  let T^(0) = (t_ij^(0)) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          if i == j or (i, j) ∈ G.E
6              t_ij^(0) = 1
7          else t_ij^(0) = 0
8  for k = 1 to n
9      let T^(k) = (t_ij^(k)) be a new n × n matrix
10     for i = 1 to n
11         for j = 1 to n
12             t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
13 return T^(n)
```

Initialise $T^{(0)}$ so that $t_{i,j}^{(0)} = 1 \iff w_{i,j} < \infty$

Same as Floyd-Warshall but replace min with ∨, and + with ∧

this algorithm uses the principles of the Floyd Warshall algorithm, but instead of calculating the distance, it simply checks if there is aa path between the elements.

3-7 initializes the matrix so all edges are 1 and vertices without connection is 0. We then run a slightly altered Floyd Warshall in lines 8 to 12.

General good to knows:

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	θ(1)	θ(n)	θ(n)	θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	θ(1)	θ(1)	θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	θ(log(n))	θ(log(n))	θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	θ(log(n))	θ(log(n))	θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

- Names for order of growth for classes of algorithms:

constant	$\theta(n^0) = \theta(1)$
logarithmic	$\theta(\lg n)$
linear	$\theta(n)$
<"en log en">	$\theta(n \lg n)$
quadratic	$\theta(n^2)$
cubic	$\theta(n^3)$
polynomial	$\theta(n^k)$, $k \geq 1$
exponential	$\theta(a^n)$, $a > 1$

Growth Rate Increasing ↓

Math

Summation rules

Powers and logarithm of arithmetic progressions [edit]

$\sum_{i=1}^{n} c = nc$ for every c that does not depend on i

$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ (Sum of the simplest arithmetic progression, consisting of the n first natural numbers.)[2][*full citation needed*]

$\sum_{i=1}^{n} (2i-1) = n^2$ (Sum of first odd natural numbers)

$\sum_{i=0}^{n} 2i = n(n+1)$ (Sum of first even natural numbers)

$\sum_{i=1}^{n} \log i = \log n!$ (A sum of logarithms is the logarithm of the product)

$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$ (Sum of the first squares, see square pyramidal number.) [2]

$\sum_{i=0}^{n} i^3 = \left(\sum_{i=0}^{n} i\right)^2 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4}$ (Nicomachus's theorem) [2]

More generally,

$$\sum_{i=0}^{n} i^p = \frac{(n+1)^{p+1}}{p+1} + \sum_{k=1}^{p} \frac{B_k}{p-k+1} \binom{p}{k} (n+1)^{p-k+1},$$

where B_k denotes a Bernoulli number (that is Faulhaber's formula).

Summation index in exponents [edit]

In the following summations, a is assumed to be different from 1.

$\sum_{i=0}^{n-1} a^i = \frac{1-a^n}{1-a}$ (sum of a geometric progression)

$\sum_{i=0}^{n-1} \frac{1}{2^i} = 2 - \frac{1}{2^{n-1}}$ (special case for $a = 1/2$)

$\sum_{i=0}^{n-1} i a^i = \frac{a - na^n + (n-1)a^{n+1}}{(1-a)^2}$ (a times the derivative with respect to a of the geometric progression)

$$\begin{aligned}\sum_{i=0}^{n-1} (b+id)\, a^i &= b\sum_{i=0}^{n-1} a^i + d\sum_{i=0}^{n-1} i a^i \\ &= b\left(\frac{1-a^n}{1-a}\right) + d\left(\frac{a - na^n + (n-1)a^{n+1}}{(1-a)^2}\right) \\ &= \frac{b(1-a^n) - (n-1)da^n}{1-a} + \frac{da(1-a^{n-1})}{(1-a)^2}\end{aligned}$$

(sum of an arithmetico–geometric sequence)

Growth rates [edit]

The following are useful approximations (using theta notation):

$$\sum_{i=1}^{n} i^c \in \Theta(n^{c+1})$$ for real c greater than −1

$$\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log_e n)$$ (See Harmonic number)

$$\sum_{i=1}^{n} c^i \in \Theta(c^n)$$ for real c greater than 1

$$\sum_{i=1}^{n} \log(i)^c \in \Theta(n \cdot \log(n)^c)$$ for non-negative real c

$$\sum_{i=1}^{n} \log(i)^c \cdot i^d \in \Theta(n^{d+1} \cdot \log(n)^c)$$ for non-negative real c, d

$$\sum_{i=1}^{n} \log(i)^c \cdot i^d \cdot b^i \in \Theta(n^d \cdot \log(n)^c \cdot b^n)$$ for non-negative real $b > 1$, c, d

Logarithm rules

Rules of Logarithms

Rule 1: $\log_b(M \cdot N) = \log_b M + \log_b N$

Rule 2: $\log_b\left(\frac{M}{N}\right) = \log_b M - \log_b N$

Rule 3: $\log_b\left(M^k\right) = k \cdot \log_b M$

Rule 4: $\log_b(1) = 0$

Rule 5: $\log_b(b) = 1$

Rule 6: $\log_b\left(b^k\right) = k$

Rule 7: $b^{\log_b(k)} = k$