ATLAS.ti Report

XOP\_Encoding\_Proj

Quotations grouped by Documents

Report created by dyg on Oct 12, 2017

# icon 2 AVT01.pdf

63 Quotations:

## icon 2:1 Binary search trees are an excellent data structure to implement associa- tive arrays, maps, sets, a…

Codings:

● Data Structure

Content:

Binary search trees are an excellent data structure to implement associa- tive arrays, maps, sets, and similar interfaces.

## icon 2:2 The main difficulty, as dis- cussed in last lecture, is that they are efficient only when they are b…

Codings:

● ->  
● Problem

Content:

The main difficulty, as dis- cussed in last lecture, is that they are efficient only when they are balanced.

## icon 2:3 Straightforward sequences of insertions can lead to highly unbalanced trees with poor asymptotic com…

Codings:

● Description

Content:

Straightforward sequences of insertions can lead to highly unbalanced trees with poor asymptotic complexity and unacceptable practical efficiency.

## icon 2:4 For example, if we insert n elements with keys that are in strictly increasing or decreasing order,…

Codings:

● Example

Content:

For example, if we insert n elements with keys that are in strictly increasing or decreasing order, the complexity will be O(n2).

## icon 2:5 On the other hand, if we can keep the height to O(log(n)), as it is for a perfectly balanced tree, t…

Codings:

● Proposal

Content:

On the other hand, if we can keep the height to O(log(n)), as it is for a perfectly balanced tree, then the commplexity is bounded by O(n ∗ log(n)).

## icon 2:6 The solution is to dynamically rebalance the search tree during insert or search operations.

Codings:

● ->  
● In vivo term introduction  
● Solution

Content:

The solution is to dynamically rebalance the search tree during insert or search operations.

## icon 2:7 We have to be careful not to destroy the ordering invariant of the tree while we rebalance.

Codings:

● ->  
● Description  
● Property

Content:

We have to be careful not to destroy the ordering invariant of the tree while we rebalance.

## icon 2:8 Because of the importance of bi- nary search trees, researchers have developed many different algori…

Codings:

● <->  
● Class  
● Meta

Content:

Because of the importance of bi- nary search trees, researchers have developed many different algorithms for keeping trees in balance, such as AVL trees, red/black trees, splay trees, or randomized binary search trees. They differ in the invariants they main- tain (in addition to the ordering invariant), and when and how the rebal- ancing is done.

## icon 2:9 In this lecture we use AVL trees, which is a simple and efficient data structure to maintain balance…

Codings:

● Data Structure

Content:

In this lecture we use AVL trees, which is a simple and efficient data structure to maintain balance, and is also the first that has been proposed.

## icon 2:10 It is named after its inventors, G.M. Adelson-Velskii and E.M. Landis, who described it in 1962.

Codings:

● ->  
● Description  
● History

Content:

It is named after its inventors, G.M. Adelson-Velskii and E.M. Landis, who described it in 1962.

## icon 2:11 Recall the ordering invariant

Codings:

● <->  
● Property  
● Review

Content:

Recall the ordering invariant

## icon 2:12 At any node with key k in a binary search tree, all keys of the elements in the left subtree are str…

Codings:

● Definition  
● Mathematic

Content:

At any node with key k in a binary search tree, all keys of the elements in the left subtree are strictly less than k, while all keys of the elements in the right subtree are strictly greater than k.

## icon 2:13 To describe AVL trees we need the concept of tree height, which we de- fine as the maximal length of…

Codings:

● <-  
● Definition  
● In vivo term introduction  
● Mathematic

Content:

To describe AVL trees we need the concept of tree height, which we de- fine as the maximal length of a path from the root to a leaf. So the empty tree has height 0, the tree with one node has height 1, a balanced tree with three nodes has height 2. If we add one more node to this last tree is will have height 3. Alternatively, we can define it recursively by saying that the empty tree has height 0, and the height of any node is one greater than the maximal height of its two children.

## icon 2:16 AVL trees maintain a height invariant (also sometimes called a balance invariant).

Codings:

● <-  
● In vivo term introduction

Content:

AVL trees maintain a height invariant (also sometimes called a balance invariant).

## icon 2:17 At any node in the tree, the heights of the left and right subtrees differs by at most 1.

Codings:

● ->  
● Definition  
● Property

Content:

At any node in the tree, the heights of the left and right subtrees differs by at most 1.

## icon 2:18 As an example, consider the following binary search tree of height 3.

Codings:

● Cartoon  
● Example

Content:

As an example, consider the following binary search tree of height 3.

## icon 2:19 If we insert a new element with a key of 14, the insertion algorithm for binary search trees without…

Codings:

● ->  
● Cartoon  
● Example  
● Operation

Content:

If we insert a new element with a key of 14, the insertion algorithm for binary search trees without rebalancing will put it to the right of 13.

## icon 2:20 Now the tree has height 4, and one path is longer than the others. However, it is easy to check that…

Codings:

● <->  
● Observation  
● State

Content:

Now the tree has height 4, and one path is longer than the others. However, it is easy to check that at each node, the height of the left and right subtrees still differ only by one.

## icon 2:21 For example, at the node with key 16, the left subtree has height 2 and the right subtree has height…

Codings:

● Cartoon  
● Observation

Content:

For example, at the node with key 16, the left subtree has height 2 and the right subtree has height 1, which still obeys our height invariant.

## icon 2:22 Now consider another insertion, this time of an element with key 15.

Codings:

● <->  
● Example  
● Operation

Content:

Now consider another insertion, this time of an element with key 15.

## icon 2:23 This is inserted to the right of the node with key 14.

Codings:

● Observation

Content:

This is inserted to the right of the node with key 14.

## icon 2:24 All is well at the node labeled 14: the left subtree has height 0 while the right subtree has height…

Codings:

● <->  
● Observation  
● State

Content:

All is well at the node labeled 14: the left subtree has height 0 while the right subtree has height 1. However, at the node labeled 13, the left subtree has height 0, while the right subtree has height 2, violating our invariant. Moreover, at the node with key 16, the left subtree has height 3 while the right subtree has height 1, also a difference of 2 and therefore an invariant violation.

## icon 2:27 We therefore have to take steps to rebalance tree.

Codings:

● Conclusion

Content:

We therefore have to take steps to rebalance tree.

## icon 2:28 We can see without too much trouble, that we can restore the height invariant if we move the node la…

Codings:

● Cartoon  
● Example  
● Proposal

Content:

We can see without too much trouble, that we can restore the height invariant if we move the node labeled 14 up and push node 13 down and to the right, resulting in the following tree.

## icon 2:29 The question is

Codings:

● <->  
● Problem

Content:

The question is

## icon 2:30 In order to understand this we need a fundamental operation called a rotation, which comes in two fo…

Codings:

● In vivo term introduction

Content:

In order to understand this we need a fundamental operation called a rotation, which comes in two forms, left rotation and right rotation.

## icon 2:31 Below, we show the situation before a left rotation.

Codings:

● <->  
● Cartoon  
● Example  
● Operation

Content:

Below, we show the situation before a left rotation.

## icon 2:32 We have generically denoted the crucial key values in question with x and y. Also, we have summarize…

Codings:

● Legend

Content:

We have generically denoted the crucial key values in question with x and y. Also, we have

summarized whole subtrees with the intervals bounding their key values. Even though we wrote −∞ and +∞, when the whole tree is a subtree of a larger tree these bounds will be generic bounds α which is smaller than x

## icon 2:33 The tree on the right is after the left rotation.

Codings:

● Cartoon  
● Description

Content:

The tree on the right is after the left rotation.

## icon 2:34 From the intervals we can see that the ordering invariants are preserved, as are the contents of the…

Codings:

● ->  
● Observation  
● State

Content:

From the intervals we can see that the ordering invariants are preserved, as are the contents of the tree. We can also see that it shifts some nodes from the right subtree to the left subtree.

## icon 2:37 We implement this with some straightforward code. First, recall the type of trees from last lecture.…

Codings:

● <-  
● Example  
● Review

Content:

We implement this with some straightforward code. First, recall the type of trees from last lecture. We do not repeat the function is\_ordtree that checks if a tree is ordered.

## icon 2:38 struct tree { elem data; struct tree\* left; struct tree\* right; }; typedef struct tree\* tree;…

Codings:

● Code  
● Definition

Content:

struct tree {

elem data;

struct tree\* left;

struct tree\* right;

};

typedef struct tree\* tree;

bool is\_ordtree(tree T);

## icon 2:39 The main point to keep in mind is to use (or save) a component of the input before writing to it. We…

Codings:

● Review

Content:

The main point to keep in mind is to use (or save) a component of the input before writing to it. We apply this idea systematically, writing to a location immediately after using it on the previous line. We repeat the type specification of tree from last lecture.

## icon 2:40 tree rotate\_left(tree T) //@requires is\_ordtree(T); //@requires T != NULL && T->right != NULL; //@en…

Codings:

● Code  
● Review

Content:

tree rotate\_left(tree T)

//@requires is\_ordtree(T);

//@requires T != NULL && T->right != NULL;

//@ensures is\_ordtree(\result);

//@ensures \result != NULL && \result->left != NULL;

{

tree root = T->right;

## icon 2:41 T->right = root->left; root->left = T; return root; }

Codings:

● Code  
● Review

Content:

T->right = root->left;

root->left = T;

return root;

}

## icon 2:42 The right rotation is entirely symmetric. First in pictures:

Codings:

● Cartoon  
● Example

Content:

The right rotation is entirely symmetric. First in pictures:

## icon 2:43 tree rotate\_right(tree T) //@requires is\_ordtree(T); //@requires T != NULL && T->left != NULL; //@en…

Codings:

● Code  
● Example

Content:

tree rotate\_right(tree T)

//@requires is\_ordtree(T);

//@requires T != NULL && T->left != NULL;

//@ensures is\_ordtree(\result);

//@ensures \result != NULL && \result->right != NULL;

{

tree root = T->left;

}

Comment:

Note that the point of this code is that it represents the same, exact, sub operation shown in the previous cartoon.

## icon 2:44 Searching for a key in an AVL tree is identical to searching for it in a plain binary search tree as…

Codings:

● <->  
● Description  
● Operation  
● Related  
● Review

Content:

Searching for a key in an AVL tree is identical to searching for it in a plain binary search tree as described in Lecture 17. The reason is that we only need the ordering invariant to find the element; the height invariant is only relevant for inserting an element.

Comment:

This is really a description, of an element, of a mandatory set of sub operations that a tree must support in order to be a data structure.

## icon 2:45 The basic recursive structure of inserting an element is the same as for searching for an element.

Codings:

● <->  
● Operation

Content:

The basic recursive structure of inserting an element is the same as for searching for an element.

## icon 2:46 We compare the element’s key with the keys associated with the nodes of the trees, inserting recursi…

Codings:

● Description

Content:

We compare the element’s key with the keys associated with the nodes of the trees, inserting recursively into the left or right subtree. When we find an element with the exact key we overwrite the element in that node. If we encounter a null tree, we construct a new tree with the element to be inserted and no children and then return it. As we return the new subtrees (with the inserted element) towards the root, we check if we violate the height invariant. If so, we rebalance to restore the invariant and then continue up the tree to the root.

Comment:

Description being solely manifest through text, no math or code.

## icon 2:47 The main cleverness of the algorithm lies in analyzing the situations when we have to rebalance and…

Codings:

● ->  
● Advantages

Content:

The main cleverness of the algorithm lies in analyzing the situations when we have to rebalance and applying the appropriate rotations to re- store the height invariant. It turns out that one or two rotations on the whole tree always suffice for each insert operation, which is a very elegant result.

## icon 2:48 First, we keep in mind that the left and right subtrees’ heights before the insertion can differ by…

Codings:

● <-  
● Description

Content:

First, we keep in mind that the left and right subtrees’ heights before the insertion can differ by at most one. Once we insert an element into one of the subtrees, they can differ by at most two. We now draw the trees in such a way that the height of a node is indicated by the height that we are drawing it at.

## icon 2:49 The first situation we describe is where we insert into the right subtree, which is already of heigh…

Codings:

● ->  
● Cartoon  
● Example  
● Observation  
● State

Content:

The first situation we describe is where we insert into the right subtree, which is already of height h + 1 where the left subtree has height h. If we are unlucky, the result of inserting into the right subtree will give us a new right subtree of height h + 2 which raises the height of the overall tree to h + 3, violating the height invariant. In the new right subtree has height h+2, either its right or the left subtree must be of height h+1 (and only one of them; think about why). If it is the right subtree we are in the situation depicted below on the left.

## icon 2:50 left rotation, the result of which is displayed to the right. In the second case we consider we once…

Codings:

● ->  
● Cartoon  
● Example  
● In vivo term introduction  
● Operation

Content:

left rotation, the result of which is displayed to the right. In the second case we consider we once again insert into the right sub-

tree, but now the left subtree of the right subtree has height h + 1.

(‐∞, +∞) x

h

In that case, a left rotation alone will not restore the invariant (see Exer- cise 1). Instead, we apply a so-called double rotation: first a right rotation at z, then a left rotation at the root. When we do this we obtain the picture on the right, restoring the height invariant.

There are two additional symmetric cases to consider, if we insert the new element on the left (see Exercise 4).

## icon 2:51 The interface for the implementation is exactly the same as for binary search trees, as is the code…

Codings:

● Description  
● Operation  
● Related

Content:

The interface for the implementation is exactly the same as for binary search trees, as is the code for searching for a key.

## icon 2:52 In various places in the algo- rithm we have to compute the height of the tree. This could be an ope…

Codings:

● ->  
● Description  
● Design

Content:

In various places in the algo- rithm we have to compute the height of the tree. This could be an operation of asymptotic complexity O(n), unless we store it in each node and just look it up. So we have:

## icon 2:53 struct tree { elem data; int height; struct tree\* left; struct tree\* right; };

Codings:

● Code

Content:

struct tree {

elem data;

int height;

struct tree\* left;

struct tree\* right;

};

## icon 2:54 typedef struct tree\* tree; /\* height(T) returns the precomputed height of T in O(1) \*/ int height(tr…

Codings:

● Code  
● Description

Content:

typedef struct tree\* tree;

/\* height(T) returns the precomputed height of T in O(1) \*/

int height(tree T) {

return T == NULL ? 0 : T->height;

}

## icon 2:55 When checking if a tree is balanced, we also check that all the heights that have been computed are…

Codings:

● <-  
● Description

Content:

When checking if a tree is balanced, we also check that all the heights that have been computed are correct.

## icon 2:56 bool is\_balanced(tree T) { if (T == NULL) return true; int h = T->height; int hl = height(T->l…

Codings:

● Code  
● Definition

Content:

bool is\_balanced(tree T) {

if (T == NULL) return true;

int h = T->height;

int hl = height(T->left);

int hr = height(T->right);

if (!(h == (hl > hr ? hl+1 : hr+1))) return false;

if (hl > hr+1 || hr > hl+1) return false;

return is\_balanced(T->left) && is\_balanced(T->right);

}

## icon 2:57 A tree is an AVL tree if it is both ordered (as defined and implementa- tion in the last lecture) an…

Codings:

● Code  
● Definition

Content:

A tree is an AVL tree if it is both ordered (as defined and implementa- tion in the last lecture) and balanced.

bool is\_avl(tree T) {

return is\_ordtree(T) && is\_balanced(T);

}

## icon 2:58 We use this, for example, in a utility function

Codings:

● ->  
● Example  
● Operation

Content:

We use this, for example, in a utility function

## icon 2:59 tree leaf(elem e) //@requires e != NULL; //@ensures is\_avl(\result); { tree T = alloc(struct tree)…

Codings:

● <-  
● Code  
● Definition

Content:

tree leaf(elem e)

//@requires e != NULL;

//@ensures is\_avl(\result);

{

tree T = alloc(struct tree);

T->data = e;

T->height = 1;

T->left = NULL;

T->right = NULL;

return T; }

## icon 2:60 The code for inserting an element into the tree is mostly identical with the code for plain binary s…

Codings:

● Operation  
● Related

Content:

The code for inserting an element into the tree is mostly identical with the code for plain binary search trees.

## icon 2:61 The difference is that after we in- sert into the left or right subtree, we call a function rebalanc…

Codings:

● Description

Content:

The difference is that after we in- sert into the left or right subtree, we call a function rebalance\_left or rebalance\_right, respectively, to restore the invariant if necessary and cal- culate the new height.

Comment:

I’m starting to think that Operation Description should be separated into operation, and description, much like definition, and mathematical

## icon 2:62 tree tree\_insert(tree T, elem e) //@requires is\_avl(T); //@ensures is\_avl(\result); { assert(e !=…

Codings:

● Code  
● Description

Content:

tree tree\_insert(tree T, elem e)

//@requires is\_avl(T);

//@ensures is\_avl(\result);

{

assert(e != NULL); /\* cannot insert NULL element \*/

if (T == NULL) {

T = leaf(e); /\* create new leaf with data e \*/

} else {

int r = compare(elem\_key(e), elem\_key(T->data));

if (r < 0) {

T->left = tree\_insert(T->left, e);

T = rebalance\_left(T); /\* also fixes height \*/

} else if (r == 0) {

T->data = e;

} else { //@assert r > 0;

T->right = tree\_insert(T->right, e);

T = rebalance\_right(T); /\* also fixes height \*/

}

}

return T; }

## icon 2:63 tree rebalance\_right(tree T) //@requires T != NULL; //@requires is\_avl(T->left) && is\_avl(T->right);…

Codings:

● Code  
● Description

Content:

tree rebalance\_right(tree T)

//@requires T != NULL;

//@requires is\_avl(T->left) && is\_avl(T->right);

/\* also requires that T->right is result of insert into T \*/

//@ensures is\_avl(\result);

{

tree l = T->left;

tree r = T->right;

int hl = height(l);

int hr = height(r);

if (hr > hl+1) {

//@assert hr == hl+2;

if (height(r->right) > height(r->left)) {

//@assert height(r->right) == hl+1;

T = rotate\_left(T);

//@assert height(T) == hl+2;

return T;

} else {

//@assert height(r->left) == hl+1;

/\* double rotate left \*/

T->right = rotate\_right(T->right);

T = rotate\_left(T);

//@assert height(T) == hl+2;

return T;

}

} else { //@assert !(hr > hl+1);

fix\_height(T);

return T; }

}

## icon 2:64 binary search trees.

Codings:

● <->  
● Data Structure  
● Related

Content:

binary search trees.

## icon 2:65 how to do this in general

Codings:

● <-

Content:

how to do this in general

## icon 2:66 We would invoke this operation if the invariants told us that we have to rebalance from right to lef…

Codings:

● <->  
● Application  
● Description

Content:

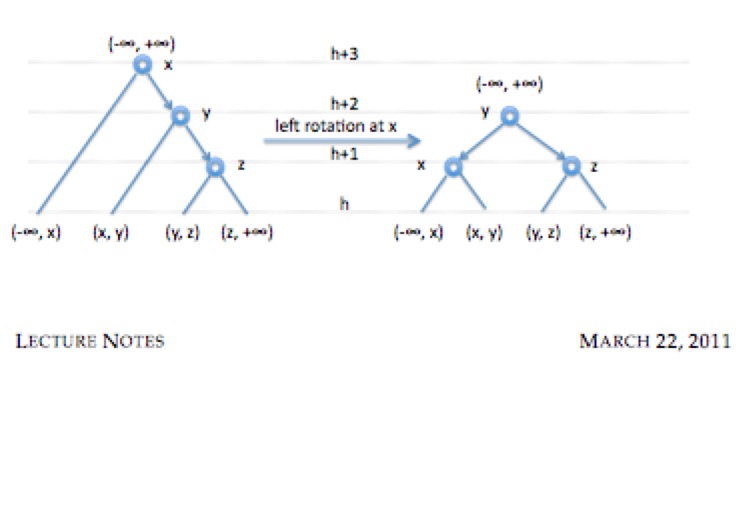
We would invoke this operation if the invariants told us that we have to rebalance from right to left.

## icon 2:67 Quotation 2:67

Codings:

● ->  
● Problem

Content:



## icon 2:68 We fix this with a left rotation,

Codings:

● ->  
● Solution

Content:

We fix this with a left rotation,

## icon 2:69 that creates a new leaf from an element (which may not be null).

Codings:

● ->  
● Application  
● Description

Content:

that creates a new leaf from an element (which may not be null).

# icon 5 AVT02.pdf

18 Quotations:

## icon 5:1 we can never be sure how balanced the tree will be, since the order of insertions will determine thi…

Codings:

● ->  
● Problem

Content:

we can never be sure how balanced the tree will be, since the order of insertions will determine this.

## icon 5:2 A solution is to create balanced BST’s as we do the insertions. An AVL tree is such a tree.

Codings:

● ->  
● Description  
● Solution

Content:

A solution is to create balanced BST’s as we do the insertions. An AVL tree is such a tree.

## icon 5:3 alance condition is this: In an AVL tree, the height of the left and right subtrees of the root diff…

Codings:

● <->  
● Definition  
● Property

Content:

alance condition is this: In an AVL tree, the height of the left and right subtrees of the root differ by at most 1, and in which the left and right subtrees are AVL trees also. This balance condiiton insures that searches, inseretions and deletetion will be close to O(Log2N), as in a fully balanced BST.

## icon 5:4 Proposition: The height of an AVL tree T storing n keys is O(log n).

Codings:

● <->  
● Property  
● Proposal

Content:

Proposition: The height of an AVL tree T storing n keys is O(log n).

## icon 5:5 Justification: The easiest way to approach this problem is to try to find the minimum number of inte…

Codings:

● ->  
● Description  
● Problem

Content:

Justification: The easiest way to approach this problem is to try to find the minimum number of internal

nodesofanAVLtreeofheighth: N(h)

## icon 5:6 ItisobviousthatthatN(1)=1andN(2)=2

Codings:

● Cases  
● Mathematic

Content:

ItisobviousthatthatN(1)=1andN(2)=2

## icon 5:7 Forh≥3,anAVL tree of height h with N(h) minimal internal nodes contains the root node, one AVL subtr…

Codings:

● Cases  
● Description  
● Mathematic

Content:

Forh≥3,anAVL tree of height h with N(h) minimal internal nodes contains the root node, one AVL subtree of height h-1 and the other AVL subtree of height h − 2.

## icon 5:8 N (h) = Since: N(h−1) > N (h) > N (h) > ... N (h) > 1+N(h−1)+N(h−2) (1) N(h−2) (2) 2N (h − 2) (subst…

Codings:

● ->  
● Derivation  
● Mathematic  
● Solution

Content:

N (h) = Since: N(h−1) > N (h) > N (h) >

... N (h) >

1+N(h−1)+N(h−2) (1) N(h−2) (2) 2N (h − 2) (substitute in (1) above, drop the + 1 term) (3

## icon 5:9 3 AVL Rotation Templates

Codings:

● Cartoon  
● Example  
● Operation

Content:

3 AVL Rotation Templates

## icon 5:10 Here is the Java code from the Weiss textbook for AVL insertion

Codings:

● Code  
● Data Structure  
● Description

Content:

Here is the Java code from the Weiss textbook for AVL insertion

## icon 5:11 AVL Tree Example:

Codings:

● Cartoon  
● Example

Content:

AVL Tree Example:

## icon 5:12 When we do an insert into a Binary Search Tree (BST)

Codings:

● Data Structure

Content:

When we do an insert into a Binary Search Tree (BST)

## icon 5:13 AVL trees keep an additional piece of information at each node: the height of each nodes left and ri…

Codings:

● <->  
● Constituent  
● Description

Content:

AVL trees keep an additional piece of information at each node: the height of each nodes left and right subtrees.

## icon 5:14 Since these heights can only differ by 1, any insertion into the tree will be analyzed to see if it…

Codings:

● <-  
● Description

Content:

Since these heights can only differ by 1, any insertion into the tree will be analyzed to see if it violates the balance condition.

## icon 5:15 If it does, then we need to reorganize the tree by a series of rotations. Rotations are simply point…

Codings:

● <->  
● Description  
● In vivo term introduction  
● Operation

Content:

If it does, then we need to reorganize the tree by a series of rotations. Rotations are simply pointer changes that rearrange the structure of the tree to keep the AVL balance condition.

## icon 5:16 The Weiss textbook has some good examples on rotations that explain how they work.

Codings:

● Aside

Content:

The Weiss textbook has some good examples on rotations that explain how they work.

## icon 5:17 ItisobviousthatthatN(1)=1andN(2)=2.

Content:

ItisobviousthatthatN(1)=1andN(2)=2.

## icon 5:18 The

Codings:

● <-

Content:

The

# icon 8 AVT03.pdf

35 Quotations:

## icon 8:1 Lecture 6: Balanced Binary Search Trees

Codings:

● Data Structure

Content:

Lecture 6: Balanced Binary Search Trees

## icon 8:2 Lecture Overview

Codings:

● Outline

Content:

Lecture Overview

## icon 8:3 Recall: Binary Search Trees (BSTs)

Codings:

● Data Structure  
● Review

Content:

Recall: Binary Search Trees (BSTs)

## icon 8:4 • rooted binary tree • each node has

Codings:

● ->  
● Constituent

Content:

• rooted binary tree

• each node has

## icon 8:5 – key – left pointer – right pointer – parent pointer

Codings:

● Definition

Content:

– key

– left pointer

– right pointer – parent pointer

## icon 8:6 See Fig.

Codings:

● Cartoon  
● Example

Content:

See Fig.

## icon 8:7 • BST property (see Fig.

Codings:

● <->  
● Property

Content:

• BST property (see Fig.

## icon 8:8 • height of node = length (# edges) of longest downward path to a leaf (see CLRS B.5

Codings:

● Definition

Content:

• height of node = length (# edges) of longest downward path to a leaf (see CLRS B.5

## icon 8:9 • BSTs support insert, delete, min, max, next-larger, next-smaller, etc. in O(h) time, where h = hei…

Codings:

● ->  
● Complexity  
● Description

Content:

• BSTs support insert, delete, min, max, next-larger, next-smaller, etc. in O(h) time, where h = height of tree (= height of root).

## icon 8:10 • h is between lgn and n: Fig.

Codings:

● Cartoon  
● Description

Content:

• h is between lgn and n: Fig.

## icon 8:11 • balanced BST maintains h = O(lg n) ⇒ all operations run in O(lg n) time.

Codings:

● Conclusion  
● Definition  
● In vivo term introduction

Content:

• balanced BST maintains h = O(lg n) ⇒ all operations run in O(lg n) time.

## icon 8:13 For every node, require heights of left & right children to differ by at most ±1.

Codings:

● Definition

Content:

For every node, require heights of left & right children to differ by at most ±1.

## icon 8:14 • treat nil tree as height -1 • each node stores its height (DATA STRUCTURE AUGMENTATION) (like subt…

Codings:

● Constituent  
● Description

Content:

• treat nil tree as height -1

• each node stores its height (DATA STRUCTURE AUGMENTATION) (like subtree

size) (alternatively, can just store difference in heights)

## icon 8:15 Fig

Codings:

● Cartoon

Content:

Fig

## icon 8:16 Balance:

Codings:

● Property

Content:

Balance:

## icon 8:17 Worst when every node differs by 1 — let Nh = (min.) # nodes in height-h AVL tree =⇒Nh =Nh−1+Nh−2+1

Codings:

● Derivation

Content:

Worst when every node differs by 1 — let Nh = (min.) # nodes in height-h AVL tree =⇒Nh =Nh−1+Nh−2+1

## icon 8:18 Alternatively:

Codings:

● Derivation  
● Pedagogical

Content:

Alternatively:

## icon 8:19 AVL Insert:

Codings:

● Operation

Content:

AVL Insert:

## icon 8:20 1. insert as in simple BST 2. work your way up tree, restoring AVL property (and updating heights a

Codings:

● Description

Content:

1. insert as in simple BST

2. work your way up tree, restoring AVL property (and updating heights a

## icon 8:21 • suppose x is lowest node violating AVL • assume x is right-heavy (left case symmetric) • if x’s ri…

Codings:

● Description

Content:

• suppose x is lowest node violating AVL

• assume x is right-heavy (left case symmetric)

• if x’s right child is right-heavy or balanced: f

## icon 8:22 follow steps in Fig. 5

Codings:

● Cartoon  
● Description

Content:

follow steps in Fig. 5

## icon 8:23 steps in Fig. 6

Codings:

● Cartoon  
● Description

Content:

steps in Fig. 6

## icon 8:24 • then continue up to x’s grandparent, greatgrandparent . . .

Codings:

● Description

Content:

• then continue up to x’s grandparent, greatgrandparent . . .

## icon 8:25 Example: An example implementation of the AVL Insert process is illustrated in Fig.

Codings:

● Cartoon  
● Example

Content:

Example: An example implementation of the AVL Insert process is illustrated in Fig.

## icon 8:26 In general, process may need several rotations before done with an Insert.

Codings:

● caveat

Content:

In general, process may need several rotations before done with an Insert.

## icon 8:27 Delete(-min) is similar — harder but possible.

Codings:

● caveat

Content:

Delete(-min) is similar — harder but possible.

## icon 8:28 AVL sort:

Codings:

● Application

Content:

AVL sort:

## icon 8:29 • insert each item into AVL tree • in-order traversal

Codings:

● ->  
● Complexity  
● Description

Content:

• insert each item into AVL tree • in-order traversal

## icon 8:31 There are many balanced search trees.

Codings:

● Class

Content:

There are many balanced search trees.

## icon 8:32 Adel’son-Velsii and Landis 1962 Bayer and McCreight 1972 (see CLRS 18) Nievergelt and Reingold 1973…

Codings:

● Description

Content:

Adel’son-Velsii and Landis 1962

Bayer and McCreight 1972 (see CLRS 18) Nievergelt and Reingold 1973

CLRS Chapter 13

Sleator and Tarjan 1985

Pugh 1989

Galperin and Rivest 1993

Seidel and Aragon 1996

## icon 8:34 Abstract Data Type(ADT): interface spec. vs.

Codings:

● Definition  
● In vivo term introduction

Content:

Abstract Data Type(ADT): interface spec. vs.

## icon 8:35 Data Structure (DS): algorithm for each op.

Codings:

● Definition  
● In vivo term introduction

Content:

Data Structure (DS): algorithm for each op.

## icon 8:36 Priority Queue ADT Q = new-empty-queue() Q.insert(x) x = Q.deletemin() x = Q.findmin() heap Θ(1) Θ(l…

Codings:

● ->  
● Complexity  
● Description

Content:

Priority Queue ADT

Q = new-empty-queue() Q.insert(x)

x = Q.deletemin()

x = Q.findmin()

heap

Θ(1) Θ(lg n) Θ(lg n) Θ(1)

AVL tree

Θ(1)

Θ(lg n) Θ(lg n) Θ(lg n) → Θ(1)

## icon 8:37 Predecessor/Successor ADT S = new-empty() S.insert(x) S.delete(x) y = S.predecessor(x) → next- Θ(n)…

Codings:

● Description

Content:

Predecessor/Successor ADT

S = new-empty()

S.insert(x)

S.delete(x)

y = S.predecessor(x) → next- Θ(n) Θ(lgn) smaller

y = S.successor(x) → next-larger Θ(n) Θ(lg n)

heap

AVL tree

Θ(1) Θ(lg n) Θ(lg n)

Θ(1) Θ(lg n) Θ(lg n)

## icon 8:38 There are many possible DSs for one ADT.

Codings:

● Design

Content:

There are many possible DSs for one ADT.

# icon 11 AVT04.pdf

64 Quotations:

## icon 11:1 AVL Trees

Codings:

● Data Structure

Content:

AVL Trees

## icon 11:2 In order to have a worst case running time for insert and delete operations to be O(log n), we must…

Codings:

● Description  
● Problem

Content:

In order to have a worst case running time for insert and delete operations to be O(log n), we must make it impossible for there to be a very long path in the binary search tree.

## icon 11:3 The rst balanced binary tree is the AVL tree, named after it's inventors, Adelson-Velskii and Landi…

Codings:

● Description  
● History

Content:

The rst balanced binary tree is the AVL tree, named after it's inventors, Adelson-Velskii and Landis.

## icon 11:4 A binary search tree is an AVL tree i each node in the tree satis es the following property:

Codings:

● Data Structure  
● Definition  
● Related

Content:

A binary search tree is an AVL tree i each node in the tree satis es the following property:

## icon 11:5 The height of the left subtree can di er from the height of the right subtree by at most 1.

Codings:

● Definition  
● Property

Content:

The height of the left subtree can di er from the height of the right subtree by at most 1.

## icon 11:6 Based on this property, we can show that the height of an AVL tree is logarithmic with respect to th…

Codings:

● Description  
● Property

Content:

Based on this property, we can show that the height of an AVL tree is logarithmic with respect to the number of nodes stored in the tree.

## icon 11:7 In particular, for an AVL tree of height H, we nd that it must contain at least FH+3 -1 nodes.

Codings:

● Description  
● Mathematic

Content:

In particular, for an AVL tree of height H, we nd that it must contain at least FH+3 -1 nodes.

## icon 11:8 To prove this, notice that the number of nodes in an AVL tree is the 1 plus the number of notes in t…

Codings:

● Observation

Content:

To prove this, notice that the number of nodes in an AVL tree is the 1 plus the number of notes in the left subtree plus the number of nodes in the right subtree.

## icon 11:9 If we let SH represent the minimum number of nodes in an AVL tree with height H, we get the followin…

Codings:

● Definition  
● Mathematic

Content:

If we let SH represent the minimum number of nodes in an AVL tree with height H, we get the following recurrence relation:

## icon 11:10 We also know that S0=1 and S1=2. Now we can prove the assertion above through induction. Problem: Pr…

Codings:

● Proof

Content:

We also know that S0=1 and S1=2. Now we can prove the assertion above through induction.

Problem: Prove that SH = FH+3 -1.

We will use induction on H, the height of the AVL tree.

Base Cases H=0: LHS = 1, RHS = F3 - 1 = 2 - 1 = 1 H=1: LHS = 2, RHS = F4 - 1 = 3 - 1 = 2

Inductive hypothesis: For an arbitrary integer k <= H, assume that Sk = Fk+3 -1.

Inductive step: Under the assumption above, prove for H=k+1 that Sk+1 = Fk+1+3 -1.

Sk+1 =Sk +Sk-1 +1

= (Fk+3 -1) + (Fk+2 -1) +1, using the I.H. twice

= (Fk+3 + Fk+2) - 1

= Fk+4 -1, using the defn. of Fibonacci numbers, to complete

proof.

It can be shown through recurrence relations, that

Fn  1/5 [(1 + 5)/2]n

So now, we have the following:

Sn  1/5 [(1 + 5)/2]n+3

This says that when the height of an AVL tree is n, the minimum number of nodes it contains is 1/5 [(1 + 5)/2]n+3.

## icon 11:11 So, in order to nd the height of a tree with n nodes, we must replace Sn with n and replace n with…

Codings:

● Solicitation

Content:

So, in order to nd the height of a tree with n nodes, we must replace Sn with n and replace n with h? Why is this the case?

## icon 11:12 n  1/5 [(1 + 5)/2]h+3 n  (1.618)h h  log 1.618 n h = O(log 2 n)

Codings:

● Proof

Content:

n  1/5 [(1 + 5)/2]h+3 n  (1.618)h

h  log 1.618 n

h = O(log 2 n)

## icon 11:13 Basically whenever an insertion or deletion is done, it is possible that the new node added or taken…

Codings:

● ->  
● Description  
● Solution

Content:

Basically whenever an insertion or deletion is done, it is possible that the new node added or taken away destroys the AVL property. In these sitiuations, we have to "rework" the tree so that the binary search tree and AVL properties are satis ed.

## icon 11:14 When an imbalance is introduced to a tree, it is localized to three nodes and their four subtrees.

Codings:

● <->  
● Description  
● State

Content:

When an imbalance is introduced to a tree, it is localized to three nodes and their four subtrees.

## icon 11:15 Denote these three nodes as A, B, and C, in their inorder listing.

Codings:

● Definition

Content:

Denote these three nodes as A, B, and C, in their inorder listing.

## icon 11:16 Structurally, they may appear in various con gurations.

Codings:

● Observation

Content:

Structurally, they may appear in various con gurations.

## icon 11:17 A couple of these are listed below:

Codings:

● Cartoon

Content:

A couple of these are listed below:

## icon 11:18 Denote the four subtrees as T0, T1, T2, and T3, also listed in their inorder listing. Here is where…

Codings:

● Legend

Content:

Denote the four subtrees as T0, T1, T2, and T3, also listed in their inorder listing. Here is where these would lie in the trees drawn above:

## icon 11:19 No matter which of these structural imbalances exist, they can all be xed the same way:

Codings:

● Observation

Content:

No matter which of these structural imbalances exist, they can all be xed the same way:

## icon 11:20 Another way we can view these transformations is through two separate types or restructuring operati…

Codings:

● <->  
● In vivo term introduction  
● Operation

Content:

Another way we can view these transformations is through two separate types or restructuring operations: a single rotation and a double rotation.

## icon 11:21 Here are the four cases we will look at: 1) insertion into the left subtree of the left child of the…

Codings:

● Cases  
● Description

Content:

Here are the four cases we will look at:

1) insertion into the left subtree of the left child of the root.

2) insertion into the right subtree of the left child of the root. 3) insertion into the left subtree of the right child of the root. 4) insertion into the right subtree of the right child of the root

## icon 11:22 Technically speaking, cases 1 and 4 are symmetric as are 2 and 3.

Codings:

● Observation

Content:

Technically speaking, cases 1 and 4 are symmetric as are 2 and 3.

## icon 11:23 For cases 1 and 4, we will perform a single rotation, and for 2 and 3 we will do a double rotation.

Codings:

● Description

Content:

For cases 1 and 4, we will perform a single rotation, and for 2 and 3 we will do a double rotation.

## icon 11:24 In the pictures I have above, the left picture is case 2 of this description, and the right picture…

Codings:

● Legend

Content:

In the pictures I have above, the left picture is case 2 of this description, and the right picture is case 4.

## icon 11:25 Why is the case on the left called a double rotation? Because we can achieve it by performing two ro…

Codings:

● Description

Content:

Why is the case on the left called a double rotation? Because we can achieve it by performing two rotations on the root node:

## icon 11:26 CC /\ /\ A T3 B T3

Codings:

● Cartoon  
● Description

Content:

CC /\ /\ A T3 B T3

## icon 11:27 Insertion into an AVL Tree So, now the question is, how can we use these rotations to actually perfo…

Codings:

● Operation

Content:

Insertion into an AVL Tree

So, now the question is, how can we use these rotations to actually perform an insert on an AVL tree?

## icon 11:28 Here are the basic steps involved: 1) Do a normal binary tree insert. 2) Restoring the tree based on…

Codings:

● Description

Content:

Here are the basic steps involved:

1) Do a normal binary tree insert.

2) Restoring the tree based on this leaf node.

## icon 11:29 This restoration is more di cult than just following the steps above. Here are the steps involved in…

Codings:

● ->  
● Operation

Content:

This restoration is more di cult than just following the steps above. Here are the steps involved in the restoration of a node:

## icon 11:30 1) Calculate the heights of the left and right subtrees, use this to set the potentially new height…

Codings:

● Description

Content:

1) Calculate the heights of the left and right subtrees, use this to set the potentially new height of the node.

2) If they are within one of each other, just go up to the parent node and continue.

3) If not, then perform the appropriate restructuring described above on that particular node, THEN go to the parent node and continue.

4) Stop when you've reached the root node.

## icon 11:31 With insertion, we are guaranteed that we will at most rebalance the tree once. When we march up the…

Codings:

● Observation

Content:

With insertion, we are guaranteed that we will at most rebalance the tree once.

When we march up the tree, at each step we are updating the height of that node, if necessary. The only nodes that need to be updated are those on the ancestral "lineage" of the inserted node.

## icon 11:32 Deletion from an AVL Tree First we will do a normal binary search tree delete.

Codings:

● Operation

Content:

Deletion from an AVL Tree

First we will do a normal binary search tree delete.

## icon 11:33 Note that structurally speaking, all deletes from a binary search tree delete nodes with zero or one…

Codings:

● Aside

Content:

Note that structurally speaking, all deletes from a binary search tree delete nodes with zero or one child.

## icon 11:34 For deleted leaf nodes, clearly the heights of the children of the node do not change. Also, the hei…

Codings:

● Observation

Content:

For deleted leaf nodes, clearly the heights of the children of the node do not change. Also, the heights of the children of a deleted node with one child do not change either.

## icon 11:35 Thus, if a delete causes a violation of the AVL Tree height property, this would HAVE to occur on so…

Codings:

● Conclusion

Content:

Thus, if a delete causes a violation of the AVL Tree height property, this would HAVE to occur on some node on the path from the parent of the deleted node to the root node.

## icon 11:36 Thus, once again, as above, to restructure the tree after a delete we will call the restructure meth…

Codings:

● Description

Content:

Thus, once again, as above, to restructure the tree after a delete we will call the restructure method on the parent of the deleted node.

## icon 11:37 One thing to note: whereas in an insert there is at most one node that needs to be rebalanced, there…

Codings:

● caveat

Content:

One thing to note: whereas in an insert there is at most one node that needs to be rebalanced, there may be multiple nodes in the delete that need to be rebalanced.

## icon 11:38 Technically speaking, at any point in the restructuring algorithm ONLY one node will ever be unbalan…

Codings:

● Observation

Content:

Technically speaking, at any point in the restructuring algorithm ONLY one node will ever be unbalanced.

## icon 11:39 But, what may happen is when that node is xed, it may propagate an error to an ancestor node.

Codings:

● Observation

Content:

But, what may happen is when that node is xed, it may propagate an error to an ancestor node.

## icon 11:40 But, this is NOT a problem because our restructuring algorithm goes all the way to the root node.

Codings:

● Conclusion

Content:

But, this is NOT a problem because our restructuring algorithm goes all the way to the root node.

## icon 11:41 1) Consider inserting 46 into the following AVL Tree:

Codings:

● Example

Content:

1) Consider inserting 46 into the following AVL Tree:

## icon 11:42 Now, let's trace through the rebalancing process from this place.

Codings:

● Operation

Content:

Now, let's trace through the rebalancing process from this place.

## icon 11:43 First, we call the method on this node. Once we set its height, we check to see if the node is balan…

Codings:

● ->  
● Description  
● Observation  
● State

Content:

First, we call the method on this node. Once we set its height, we check to see if the node is balanced. (This simply looks up the heights of the left and right subtrees, and decides if the di erence is more than 1.) In this case, the node is balanced, so we march up to the parent node, that stores 44.

We will trace through the same steps here, setting the new height of this node (this is important!) and determining that this node is balanced, since its left subtree has a height of -1 and the right subtree has a height of 0.

Similarly, we set the height and decide that the nodes storing 40 and 48 are balanced as well. Finally, when we reach the root node storing 32, we realize that our tree is imbalanced.

## icon 11:44 Now, we nally get to execute the code inside the if statement in the rebalance method. Here we set…

Codings:

● <-  
● Description

Content:

Now, we nally get to execute the code inside the if statement in the rebalance method. Here we set xPos to be the tallest grandchild of the root node. (This is the node storing 40, since its height is 2.)

## icon 11:45 Thus, the restructuring occurs on the nodes containing the 32, 48 and 40.

Codings:

● Conclusion

Content:

Thus, the restructuring occurs on the nodes containing the 32, 48 and 40.

## icon 11:46 40 /\

Codings:

● Cartoon  
● Conclusion

Content:

40 /\

## icon 11:47 2) Now, for the second example, consider inserting 61 into the following AVL Tree:

Codings:

● Example

Content:

2) Now, for the second example, consider inserting 61 into the following AVL Tree:

## icon 11:48 32 /\

Codings:

● Cartoon

Content:

32 /\

## icon 11:49 61, inserted Tracing through the code, we nd the rst place an imbalance occurs tracing up the ance…

Codings:

● ->  
● Description  
● Observation  
● State

Content:

61, inserted Tracing through the code, we nd the rst place an imbalance occurs tracing up the ancestry of the node storing 61 is at the noce storing 56. This time, we have that node A stores 56, node B stores 60, and node C stores 62. Using our restucturing algorithm, we nd the tallest grandchild of 56 to be 62, and

rearrange the tree as follows:

## icon 11:50 32 /\

Codings:

● Cartoon  
● Conclusion

Content:

32 /\

## icon 11:51 T0 is the subtree rooted at 52, T1 is the subtree rooted at 58, T2 is the subtree rooted at 61, and…

Codings:

● Legend

Content:

T0 is the subtree rooted at 52, T1 is the subtree rooted at 58, T2 is the subtree rooted at 61, and T3 is a null subtree.

## icon 11:52 3) For this example, we will delete the node storing 8 from the AVL tree below:

Codings:

● <-  
● Example

Content:

3) For this example, we will delete the node storing 8 from the AVL tree below:

## icon 11:53 32 /\

Codings:

● ->  
● Cartoon  
● State

Content:

32 /\

## icon 11:54 Tracing through the code, we nd that we must rst call the rebalance method on the parent of the de…

Codings:

● Observation

Content:

Tracing through the code, we nd that we must rst call the rebalance method on the parent of the deleted node, which stores 16. This node needs rebalancing and gets restructured as follows:

## icon 11:55 32 /\

Codings:

● Cartoon  
● Conclusion

Content:

32 /\

## icon 11:56 Notice that all four subtrees for this restructuring are null, and we only use the nodes A, B, and C…

Codings:

● Observation

Content:

Notice that all four subtrees for this restructuring are null, and we only use the nodes A, B, and C. Next, we march up to the parent of the node storing 24, the node storing 32. Once again, this node is imbalanced.

## icon 11:57 The reason for this is that the restructuring of the node with a 16 reduced the height of that subtr…

Codings:

● Implication

Content:

The reason for this is that the restructuring of the node with a 16 reduced the height of that subtree. By doing so, there was in INCREASE in the di erence of height between the subtrees of the old parent of the node storing 16. This increase could propogate an imbalance in the AVL tree.

## icon 11:58 When we restructure at the node storing the 32, we identify the node storing the 56 as the tallest g…

Codings:

● Description

Content:

When we restructure at the node storing the 32, we identify the node storing the 56 as the tallest grandchild.

## icon 11:59 Following the steps we've done previously, we get the nal tree as follows:

Codings:

● Cartoon  
● Conclusion

Content:

Following the steps we've done previously, we get the nal tree as follows:

## icon 11:60 4) The nal example, we will delete the node storing 4 from the AVL tree below:

Codings:

● <-  
● Example

Content:

4) The nal example, we will delete the node storing 4 from the AVL tree below:

## icon 11:61 When we call rebalance on the node storing an 8, (the parent of the deleted node), we do NOT nd an…

Codings:

● Description

Content:

When we call rebalance on the node storing an 8, (the parent of the deleted node), we do NOT nd an imbalance at an ancestral node until we get to the root node of the tree.

## icon 11:62 Here we once again identify the node storing 32 as node A, the node storing 48 as node B and the nod…

Codings:

● Observation

Content:

Here we once again identify the node storing 32 as node A, the node storing

48 as node B and the node storing 56 as node C.

## icon 11:63 Accordingly, we restructure as follows:

Codings:

● Cartoon  
● Conclusion

Content:

Accordingly, we restructure as follows:

## icon 11:64 Now the question remains, how do we maintain an AVL tree? What extra work do we have to do to make s…

Codings:

● ->  
● Description  
● Problem

Content:

Now the question remains, how do we maintain an AVL tree? What extra work do we have to do to make sure that the AVL property is maintained?

# icon 16 AVT05.pdf

59 Quotations:

## icon 16:1 Binary Search Trees

Codings:

● Data Structure

Content:

Binary Search Trees

## icon 16:2 Binary Search Trees are not Unique The same data can yield different binary search trees

Codings:

● ->  
● Description  
● Disadvantages

Content:

Binary Search Trees are not Unique

The same data can yield different binary search trees

## icon 16:3 1,2,4,5,8

Codings:

● Cartoon  
● Example

Content:

1,2,4,5,8

## icon 16:4 BST in worst case • BST degenerated into linear sequence • expectednumberofcomparisonsis(n+1)/2

Codings:

● Description  
● Sequence

Content:

BST in worst case

• BST degenerated into linear sequence

• expectednumberofcomparisonsis(n+1)/2

## icon 16:5 Balanced BST • height is O(log2 n) • O(log2n)comparisons

Codings:

● Description  
● Implication  
● Sequence

Content:

Balanced BST

• height is O(log2 n)

• O(log2n)comparisons

## icon 16:7 • Self balancing BST/height balanced BST

Codings:

● Description  
● Sequence

Content:

• Self balancing BST/height balanced BST

## icon 16:8 • AVL = Adelson-Velskii och Landis, 1962

Codings:

● ->  
● Description  
● History

Content:

• AVL = Adelson-Velskii och Landis, 1962

## icon 16:9 • The idea: Keep updated balance information in each node

Codings:

● <-  
● Description

Content:

• The idea: Keep updated balance information in each node

## icon 16:10 • AVLpropertyForeachinternalnodevinT,theheightsofthechildrenofvdifferbyatmost1...or alternatively. .…

Codings:

● ->  
● Description  
● Property

Content:

• AVLpropertyForeachinternalnodevinT,theheightsofthechildrenofvdifferbyatmost1...or

alternatively. . . For each internal node v in T

## icon 16:11 the following holds: b(v) ∈ {−1, 0, 1}, where b(v) = height(leftChild(v)) − height(rightChild(v))

Codings:

● Definition  
● Mathematic

Content:

the following holds: b(v) ∈ {−1, 0, 1}, where

b(v) = height(leftChild(v)) − height(rightChild(v))

## icon 16:12 TheheightofanAVLtreestoringnelementsisO(logn).

Codings:

● <->  
● Definition  
● Property

Content:

TheheightofanAVLtreestoringnelementsisO(logn).

## icon 16:13 We can do find, insert, and remove in an AVL tree in time O(logn) while preserving the AVL property.

Codings:

● ->  
● Definition  
● Implication  
● Property

Content:

We can do find, insert, and remove in an AVL tree in time O(logn) while preserving the AVL property.

## icon 16:14 an AVL tree

Codings:

● <-  
● Cartoon  
● Example

Content:

an AVL tree

## icon 16:15 Insertion in AVL trees

Codings:

● ->  
● Operation

Content:

Insertion in AVL trees

## icon 16:16 • The new node changes the tree height and the tree has to be re-balanced. – Information about the h…

Codings:

● Description  
● Sequence

Content:

• The new node changes the tree height and the tree has to be re-balanced.

– Information about the height of sub trees can be represented in different ways:

∗ Store the height explicitly in each node

∗ Store the balance factor in each node

• The change is usually described as a left or right rotation of a sub tree.

• One rotation is sufficient to re-balance the tree

## icon 16:17 Insertion in AVL trees (simple cases)

Codings:

● Cartoon  
● Example

Content:

Insertion in AVL trees (simple cases)

## icon 16:18 • Start from the new node and search upwards until a node x is found, such that its grandparent z is…

Codings:

● Description

Content:

• Start from the new node and search upwards until a node x is found, such that its grandparent z is unbalanced. Mark the parent of x with y. Reconstruct the tree as follows:

## icon 16:19 – Rename x,y,z to a,b,c based on their inorder order. – Let T0,T1,T2,T3 be an enumeration in inorder…

Codings:

● Sequence

Content:

– Rename x,y,z to a,b,c based on their inorder order.

– Let T0,T1,T2,T3 be an enumeration in inorder of the sub trees of x, y och z. (None of the sub

trees can have x, y, or z as root.)

– Exchange z for b, its children are now a and c.

– T0 and T1 are children of a, and T2 and T3 are children of c.

## icon 16:20 Example: insertion in an AVL tree

Codings:

● Cartoon  
● Example

Content:

Example: insertion in an AVL tree

## icon 16:21 Four different rotations

Codings:

● ->  
● Observation  
● State

Content:

Four different rotations

## icon 16:22 If b = y we call it a single rotation.”Rotate y up above z”

Codings:

● <-  
● Description

Content:

If b = y we call it a single rotation.”Rotate y up above z”

## icon 16:23 rotations

Codings:

● ->  
● Observation  
● State

Content:

rotations

## icon 16:24 If b = y we call it a single rotation.”Rotate y up above z”

Codings:

● <-  
● Description

Content:

If b = y we call it a single rotation.”Rotate y up above z”

## icon 16:25 rotations

Codings:

● ->  
● Observation  
● State

Content:

rotations

## icon 16:26 If b = x we call it a double rotation.”Rotate x up above y and then above z”

Codings:

● <-  
● Description

Content:

If b = x we call it a double rotation.”Rotate x up above y and then above z”

## icon 16:27 Four different rotations

Codings:

● ->  
● Observation  
● State

Content:

Four different rotations

## icon 16:28 If b = x we call it a double.”Rotate x up above y and then above z”

Codings:

● <-  
● Description

Content:

If b = x we call it a double.”Rotate x up above y and then above z”

## icon 16:29 other way to describe it

Codings:

● Cartoon  
● Example

Content:

other way to describe it

## icon 16:30 Another way to describe it

Codings:

● Cartoon

Content:

Another way to describe it

## icon 16:31 . . . and then insert something which destroys it

Codings:

● Description

Content:

. . . and then insert something which destroys it

## icon 16:32 Do a single rotation

Codings:

● ->  
● Cartoon  
● Observation  
● Proposal  
● State

Content:

Do a single rotation

## icon 16:33 Do a single rotation

Codings:

● Cartoon  
● Description

Content:

Do a single rotation

## icon 16:34 Do a single rotation

Codings:

● Cartoon  
● Description

Content:

Do a single rotation

## icon 16:35 Do a single rotation

Codings:

● Cartoon  
● Description

Content:

Do a single rotation

## icon 16:36 Do a single rotation

Codings:

● Cartoon  
● Description

Content:

Do a single rotation

## icon 16:37 Done!

Codings:

● Conclusion

Content:

Done!

## icon 16:38 Another way to describe it

Codings:

● <-  
● Cartoon  
● Example

Content:

Another way to describe it

## icon 16:39 . . . this time the insertion is in another position

Codings:

● Cartoon  
● Example

Content:

. . . this time the insertion is in another position

## icon 16:40 . . . hmm, we did not get balance

Codings:

● ->  
● Observation  
● State

Content:

. . . hmm, we did not get balance

## icon 16:42 Start over. . . and look at the structure

Codings:

● <->  
● Observation  
● State

Content:

Start over. . . and look at the structure

## icon 16:43 We have to perform a double rotation

Codings:

● Proposal

Content:

We have to perform a double rotation

## icon 16:44 We have to perform a double rotation

Codings:

● Cartoon  
● Description

Content:

We have to perform a double rotation

## icon 16:45 We have to perform a double rotation

Codings:

● Cartoon  
● Description

Content:

We have to perform a double rotation

## icon 16:46 We have to perform a double rotation

Codings:

● Cartoon  
● Description

Content:

We have to perform a double rotation

## icon 16:47 We have to perform a double rotation

Codings:

● Cartoon  
● Description

Content:

We have to perform a double rotation

## icon 16:48 We have to perform a double rotation

Codings:

● Cartoon  
● Description

Content:

We have to perform a double rotation

## icon 16:49 Done!

Codings:

● Conclusion

Content:

Done!

## icon 16:50 restructuring = rotations. . .

Codings:

● Operation

Content:

restructuring = rotations. . .

## icon 16:51 Some authors use left and right rotations: Single left rotation: • left part of subtree (a and j) is…

Codings:

● Description

Content:

Some authors use left and right rotations: Single left rotation: • left part of subtree (a and j) is lowered

## icon 16:52 • we have ”rotated (up) b above a”

Codings:

● Cartoon  
● Example

Content:

• we have ”rotated (up) b above a”

## icon 16:54 Double rotations. . .

Codings:

● <->  
● Operation

Content:

Double rotations. . .

## icon 16:55 Two rotations are needed when the nodes to be re-balanced are placed in a zig-zag pattern.

Codings:

● Description

Content:

Two rotations are needed when the nodes to be re-balanced are placed in a zig-zag pattern.

## icon 16:56 • Rotate b up above a • Rotate b up above c

Codings:

● Cartoon  
● Description  
● Sequence

Content:

• Rotate b up above a

• Rotate b up above c

## icon 16:57 Deletion in an AVL tree

Codings:

● <->  
● Operation

Content:

Deletion in an AVL tree

## icon 16:58 • find and remove as in an ordinary binary search tree • Update balance information on the way back…

Codings:

● Description  
● Sequence

Content:

• find and remove as in an ordinary binary search tree

• Update balance information on the way back up to the root • Iftounbalanced:Restructure...but...

– When we restore balance in one position it might incur unbalance in another position – Have to repeat the re-balancing procedure (or balance control) until the root is reached – At most O(logn) re-balancing operations

## icon 16:59 AVL-trees TDDC32

Codings:

● Data Structure

Content:

AVL-trees TDDC32

## icon 16:60 Trees

Codings:

● <-

Content:

Trees

## icon 16:61 Tree

Codings:

● <-

Content:

Tree

## icon 16:62 Example

Codings:

● <-

Content:

Example

# icon 14 DJK01.pdf

19 Quotations:

## icon 14:1 10.2 Dijkstra's Algorithm

Codings:

● Algorithm

Content:

10.2 Dijkstra's Algorithm

## icon 14:2 Djikstra's algorithm (named after its discover, E.W. Dijkstra)

Codings:

● ->  
● Description  
● History

Content:

Djikstra's algorithm (named after its discover, E.W. Dijkstra)

## icon 14:3 solves the problem of finding the shortest path from a point in a graph (the source) to a destinatio…

Codings:

● <->  
● Description  
● Problem

Content:

solves the problem of finding the shortest path from a point in a graph (the source) to a destination.

## icon 14:4 It turns out that one can find the shortest paths from a given source to all points in a graph in th…

Codings:

● ->  
● Class  
● Description

Content:

It turns out that one can find the shortest paths from a given source to all points in a graph in the same time, hence this problem is sometimes called the single-source shortest paths problem.

## icon 14:5 The somewhat unexpected result that all the paths can be found as easily as one further demonstrates…

Codings:

● Aside

Content:

The somewhat unexpected result that all the paths can be found as easily as one further demonstrates the value of reading the literature on algorithms!

## icon 14:6 This problem is related to the spanning tree one.

Codings:

● <->  
● Problem  
● Related

Content:

This problem is related to the spanning tree one.

## icon 14:7 The graph representing all the paths from one vertex to all the others must be a spanning tree - it…

Codings:

● Description

Content:

The graph representing all the paths from one vertex to all the others must be a spanning tree - it must include all vertices. There will also be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex.

## icon 14:8 G = (V,E) where V is a set of vertices and E is a set of edges.

Codings:

● <->  
● Data Structure  
● Definition  
● Mathematic

Content:

G = (V,E) where V is a set of vertices and E is a set of edges.

## icon 14:9 Dijkstra's

Codings:

● <-

Content:

Dijkstra's

## icon 14:10 algorithm keeps two sets of vertices: S the set of vertices whose shortest paths from the source hav…

Codings:

● <->  
● Constituent  
● Definition

Content:

algorithm keeps two sets of vertices:

S the set of vertices whose shortest paths from the source have already been

determined and

V- S

The other data structures needed are:

d array of best estimates of shortest path to each vertex pi an array of predecessors for each vertex

## icon 14:11 The basic mode of operation is:

Codings:

● <-  
● Description

Content:

The basic mode of operation is:

## icon 14:12 Relaxation

Codings:

● ->  
● Operation

Content:

Relaxation

## icon 14:13 The relaxation process updates the costs of all the vertices, v, connected to a vertex, u, if we cou…

Codings:

● Description  
● In vivo term introduction  
● Mathematic

Content:

The relaxation process updates the costs of all the vertices, v, connected to a vertex, u, if we could improve the best estimate of the shortest path to v by including (u,v) in the path to v.

## icon 14:14 The relaxation procedure proceeds as follows:

Codings:

● Code  
● Definition

Content:

The relaxation procedure proceeds as follows:

## icon 14:15 This sets up the graph so that each node has no predecessor (pi[v] = nil) and the estimates of the c…

Codings:

● Description

Content:

This sets up the graph so that each node has no predecessor (pi[v] = nil) and the estimates of the cost (distance) of each node from the source (d[v]) are infinite, except for the source node itself (d[s] = 0).

## icon 14:16 Note that we have also introduced a further way to store a graph (or part of a graph - as this struc…

Codings:

● Aside  
● Description  
● Mathematic

Content:

Note that we have also introduced a further way to store a graph (or part of a graph - as this structure can only store a spanning tree), the predecessor sub-graph - the list of predecessors of each node,

pi[j], 1 <= j <= |V| The edges in the predecessor sub-graph are (pi[v],v).

## icon 14:17 The relaxation procedure checks whether the current best estimate of the shortest distance to v (d[v…

Codings:

● Description

Content:

The relaxation procedure checks whether the current best estimate of the shortest distance to v (d[v]) can be improved by going through u (i.e. by making u the predecessor of v):

## icon 14:18 relax( Node u, Node v, double w[][] )

Codings:

● Code  
● Definition

Content:

relax( Node u, Node v, double w[][] )

## icon 14:19 The algorithm itself is now:

Codings:

● <-  
● Definition

Content:

The algorithm itself is now:

# icon 3 DJK02.pdf

51 Quotations:

## icon 3:2 In recitation we talked a bit about graphs: how to represent them and how to traverse them. Today we…

Codings:

● Algorithm  
● Description

Content:

In recitation we talked a bit about graphs: how to represent them and how to traverse them. Today we will discuss one of the most important graph algorithms: Dijkstra's shortest path algorithm, a greedy algorithm that efficiently finds shortest paths in a graph.

## icon 3:3 Many more problems than you might at first think can be cast as shortest path problems, making this…

Codings:

● ->  
● Description  
● Motivation

Content:

Many more problems than you might at first think can be cast as shortest path problems, making this algorithm a powerful and general tool.

## icon 3:4 For example, Dijkstra's algorithm is a good way to implement a service like MapQuest that finds the…

Codings:

● <->  
● Application  
● Description

Content:

For example, Dijkstra's algorithm is a good way to implement a service like MapQuest that finds the shortest way to drive between two points on the map. It can also be used to solve problems like network routing, where the goal is to find the shortest path for data packets to take through a switching network. It is also used in more general search algorithms for a variety of problems ranging from automated circuit layout to speech recognition.

## icon 3:5 Let's start by defining a data abstraction for weighted, directed graphs so we can express algorithm…

Codings:

● <->  
● Data Structure  
● In vivo term introduction

Content:

Let's start by defining a data abstraction for weighted, directed graphs so we can express algorithms independently of the implementation of graphs themselves.

## icon 3:6 In a weighted graph, each of its edges has a nonnegative weight that we can think of as the distance…

Codings:

● Description

Content:

In a weighted graph, each of its edges has a nonnegative weight that we can think of as the distance one must travel when going along that edge.

## icon 3:7 (\* A signature for directed graphs. The signature is \* simplified by not explicitly representing…

Codings:

● Code  
● Definition

Content:

(\* A signature for directed graphs. The signature is

\* simplified by not explicitly representing edges as

\* type. \*)

signature WGRAPH = sig

type graph (\* A directed graph comprising a set of

\* vertices and directed edges with nonnegative

\* weights. \*)

type vertex (\* A vertex, or node, of the graph \*)

(\* Whether two vertices are the same vertex. \*)

val eq: vertex\*vertex­>bool

(\* All vertices in the graph, without any duplicates.

\* Run time: O(|V|). \*)

val vertices: graph­>vertex list

(\* outgoing(v) is a list of pairs (v\_i,w\_i), one for each

\* edge leaving the vertex v. For each index i, the

\* corresponding edge leaves v and goes to v\_i, and

\* has weight w\_i.

\* Run time is linear in the length of the result. \*)

val outgoing: vertex­>(vertex\*int) list end

## icon 3:8 There are some constraints on the running time of certain operations in this specification.

Codings:

● ->  
● caveat  
● Description  
● Operation

Content:

There are some constraints on the running time of certain operations in this specification.

## icon 3:10 structure Graph : WGRAPH = struct (\* Note: vertex must contain a ref to allow graphs \* containi…

Codings:

● Code  
● Definition

Content:

structure Graph : WGRAPH = struct

(\* Note: vertex must contain a ref to allow graphs

\* containing cycles to be built and to give vertices

\* a notion of unique identity (ref identity).

\* The type vertex must be a datatype to permit it to

\* be defined recursively. \*)

datatype vertex = V of (vertex\*int) list ref type graph = vertex list

fun eq(V(v1), V(v2)) = (v1 = v2) fun vertices(g) = g

fun outgoing(V(lr)) = !lr end

## icon 3:11 A path through the graph is a sequence (v1, ..., vn) such that the graph contains an edge e1 going f…

Codings:

● Definition  
● In vivo term introduction  
● Mathematic

Content:

A path through the graph is a sequence (v1, ..., vn) such that the graph contains an edge e1 going from

v1 to v2, an edge e2 going from v2 to v3, and so on. That is, all the edges must be traversed in the

forward direction.

## icon 3:12 The length of a path is the sum of the weights along these edges e1,..., en­1. We call this property…

Codings:

● Definition  
● In vivo term introduction

Content:

The length of a path is the sum of the weights along these edges e1,..., en­1. We call

this property "length" even though for some graphs it may represent some other quantity: for example, money or time.

## icon 3:14 Given two vertices v and v', what is the shortest path through the graph that goes from v to v' ? Th…

Codings:

● Definition  
● Problem

Content:

Given two vertices v and v', what is the shortest path through the graph that goes from v to v' ? That is, the path for which summing up the weights along all the edges from v to v' results in the smallest sum possible.

## icon 3:15 It turns out that we can solve this problem efficiently by solving a more general problem, the singl…

Codings:

● ->  
● Abstraction  
● Problem  
● Related

Content:

It turns out that we can solve this problem efficiently by solving a more general problem, the single­ source shortest­path problem:

## icon 3:16 Given a vertex v, what is the length of the shortest path from v to every vertex v' in the graph?

Codings:

● Definition  
● Mathematic

Content:

Given a vertex v, what is the length of the shortest path from v to every vertex v' in the graph?

## icon 3:18 The single­source shortest path problem can also be formulated on an undirected graph; however, it i…

Codings:

● caveat  
● In vivo term introduction

Content:

The single­source shortest path problem can also be formulated on an undirected graph; however, it is most easily solved by converting the undirected graph into a directed graph with twice as many edges, and then running the algorithm for directed graphs.

## icon 3:19 There are other shortest­path problems of interest, such as the all­pairs shortest­path problem: fin…

Codings:

● ->  
● Class  
● Description

Content:

There are other shortest­path problems of interest, such as the all­pairs shortest­path problem: find the lengths of shortest paths between all possible source–destination pairs. The Floyd­Warshall algorithm is a good way to solve this problem efficiently.

## icon 3:21 Let's consider a simpler problem: solving the single­source shortest path problem for an unweighted…

Codings:

● Description  
● Problem  
● Related

Content:

Let's consider a simpler problem: solving the single­source shortest path problem for an unweighted directed graph. In this case we are trying to find the smallest number of edges that must be traversed in order to get to every vertex in the graph. This is the same problem as solving the weighted version where all the weights happen to be 1.

## icon 3:23 Do we know an algorithm for determining this? Yes

Codings:

● ->  
● Solution

Content:

Do we know an algorithm for determining this? Yes

## icon 3:24 The running time of that algorithm is O(V+E) where V is the number of vertices and E is the number o…

Codings:

● ->  
● Complexity  
● Description

Content:

The running time of that algorithm is O(V+E) where V is the number of vertices and E is the number of edges, because it pushes each reachable vertex onto the queue and considers each outgoing edge from it once. There can't be any faster algorithm for solving this problem, because in general the algorithm must at least look at the entire graph, which has size O(V+E).

## icon 3:26 We saw in recitation that we could express both breadth­first and depth­first search with the same s…

Codings:

● <->  
● Design  
● Review

Content:

We saw in recitation that we could express both breadth­first and depth­first search with the same simple algorithm that varied just in the order in which vertices are removed from the queue. We just need an efficient implementation of sets to keep track of the vertices we have visited already. A hash table fits the bill perfectly with its O(1) amortized run time for all operations. Here is an imperative graph search algorithm that takes a source vertex v0 and performs graph search outward from it:

## icon 3:27 (\* Simple graph traversal (BFS or DFS) \*) let val q: queue = new\_queue() val visited: vertexSet = cr…

Codings:

● Code

Content:

(\* Simple graph traversal (BFS or DFS) \*)

let val q: queue = new\_queue()

val visited: vertexSet = create\_vertexSet() fun expand(v: vertex) =

let val neighbors: vertex list = Graph.outgoing(v) fun handle\_edge(v': vertex): unit =

if not (member(visited,v')) then ( add(visited, v');

push(q, v') ) else () )

in

end in

add(visited, v0);

expand(v0);

while (not (empty\_queue(q)) do expand(pop(q))

end

## icon 3:28 This code implicitly divides the set of vertices into three sets: 1. The completed vertices: visited…

Codings:

● Description  
● In vivo term introduction

Content:

This code implicitly divides the set of vertices into three sets:

1. The completed vertices: visited vertices that have already been removed from the queue. 2. The frontier: visited vertices on the queue

3. The unvisited vertices: everything else

## icon 3:29 Except for the initial vertex v0, the vertices in set 2 are always neighbors of vertices in set 1. T…

Codings:

● Description

Content:

Except for the initial vertex v0, the vertices in set 2 are always neighbors of vertices in set 1. Thus, the

queued vertices form a frontier in the graph, separating sets 1 and 3.

## icon 3:30 The expand function moves a frontier vertex into the completed set and then expands the frontier to…

Codings:

● ->  
● Description  
● Operation

Content:

The expand function moves a frontier vertex into the completed set and then expands the frontier to include any previously unseen neighbors of the new frontier vertex.

## icon 3:32 The kind of search we get from this algorithm is determined by the pop function, which selects a ver…

Codings:

● <-

Content:

The kind of search we get from this algorithm is determined by the pop function, which selects a vertex from a queue. If q is a FIFO queue, we do a breadth­first search of the graph. If q is a LIFO queue, we do a depth­first search.

## icon 3:33 If the graph is unweighted, we can use a FIFO queue and keep track of the number of edges taken to g…

Codings:

● Description

Content:

If the graph is unweighted, we can use a FIFO queue and keep track of the number of edges taken to get to a particular node. We augment the visited set to keep track of the number of edges traversed from v0; it becomes a hash table implementing a map from vertices to edge counts (ints). The only

modification needed is in expand, which adds to the frontier a newly found vertex at a distance one greater than that of its neighbor already in the frontier.

## icon 3:34 (\* unweighted single­source shortest path \*) let val q: queue = new\_queue() val visited: vertexMap =…

Codings:

● Code

Content:

(\* unweighted single­source shortest path \*)

let val q: queue = new\_queue()

val visited: vertexMap = create\_vertexMap() (\* visited maps vertex­>int \*)

fun expand(v: vertex) =

let val neighbors: vertex list = Graph.outgoing(v) val dist: int = valOf(get(visited, v))

fun handle\_edge(v': vertex) =

case get(visited, v') of

SOME(d') => () (\* d' <= dist+1 \*)

| NONE => ( add(visited, v', dist+1);

push(q, v') )

in

end in

add(visited, v0, 0);

expand(v0);

while (not (empty\_queue(q)) do expand(pop(q))

end

## icon 3:35 Now we can generalize to the problem of computing the shortest path between two vertices in a weight…

Codings:

● Abstraction  
● Problem

Content:

Now we can generalize to the problem of computing the shortest path between two vertices in a weighted graph.

## icon 3:37 As in that algorithm, we keep a visited map that maps vertices to their distances from the source ve…

Codings:

● ->  
● Description  
● Operation

Content:

As in that algorithm, we keep a visited map that maps vertices to their distances from the source vertex v0. We change expand so that Instead of adding 1 to the

distance, its adds the weight of the edge traversed. Here is a first cut at an algorithm:

## icon 3:39 let val q: queue = new\_queue() val visited: vertexMap = create\_vertexMap() fun expand(v: vertex) = l…

Codings:

● Code  
● Proposal

Content:

let val q: queue = new\_queue()

val visited: vertexMap = create\_vertexMap() fun expand(v: vertex) =

let val neighbors: vertex list = Graph.outgoing(v) val dist: int = valOf(get(visited, v))

fun handle\_edge(v': vertex, weight: int) =

case get(visited, v') of SOME(d') =>

if dist+weight < d'

then add(visited, v', dist+weight) else ()

| NONE => ( add(visited, v', dist+weight);

push(q, v') )

in

end in

add(visited, v0, 0);

expand(v0);

while (not (empty\_queue(q)) do expand(pop(q))

end

## icon 3:40 This is nearly Dijkstra's algorithm, but it doesn't work. To see why, consider the following graph,…

Codings:

● Cartoon  
● Example  
● Related

Content:

This is nearly Dijkstra's algorithm, but it doesn't work. To see why, consider the following graph, where the source vertex is v0 = A.

## icon 3:41 The first pass of the algorithm will add vertices B and D to the map visited, with distances 1 and 5…

Codings:

● ->  
● Description  
● Example  
● Problem

Content:

The first pass of the algorithm will add vertices B and D to the map visited, with distances 1 and 5 respectively. D will then become part of the completed set with distance 5. Yet there is a path from A to D with the shorter length 3.

## icon 3:42 We need two fixes to the algorithm just presented:

Codings:

● ->  
● Proposal  
● Solution

Content:

We need two fixes to the algorithm just presented:

## icon 3:43 In the SOME case a check is needed to see whether the path just discovered to the vertex v' is an im…

Codings:

● Description

Content:

In the SOME case a check is needed to see whether the path just discovered to the vertex v' is an improvement on the previously discovered path (which had length d)

## icon 3:44 2. The queue q should not be a FIFO queue. Instead, it should be a priority queue where the prioriti…

Codings:

● ->  
● Description  
● Design  
● In vivo term introduction

Content:

2. The queue q should not be a FIFO queue. Instead, it should be a priority queue where the priorities of the vertices in the queue are their distances recorded in visited. That is, pop(q) should be a priority queue extract\_min operation that removes the vertex with the smallest distance.

## icon 3:45 The priority queue must also support a new operation increase\_priority(q,v) that increases the prior…

Codings:

● <->  
● Description  
● Operation

Content:

The priority queue must also support a new operation increase\_priority(q,v) that increases the priority of an element v already in the queue q. This new operation is easily implemented for heaps using the same bubbling­up algorithm used when performing heap insertions.

## icon 3:46 With these two modifications, we have Dijkstra's algorithm:

Codings:

● Conclusion

Content:

With these two modifications, we have Dijkstra's algorithm:

## icon 3:49 Each time that expand is called, a vertex is moved from the frontier set to the completed set.

Codings:

● Description  
● Operation

Content:

Each time that expand is called, a vertex is moved from the frontier set to the completed set.

## icon 3:50 Dijkstra's algorithm is an example of a greedy algorithm, because it just chooses the closest fronti…

Codings:

● <-  
● Description  
● In vivo term introduction

Content:

Dijkstra's algorithm is an example of a greedy algorithm, because it just chooses the closest frontier vertex at every step. A locally optimal, "greedy" step turns out to produce the global optimal solution. We can see that this algorithm finds the shortest­path distances in the graph example above, because it will successively move B and C into the completed set, before D, and thus D's recorded distance has been correctly set to 3 before it is selected by the priority queue.

## icon 3:51 The algorithm works because it maintains the following two invariants:

Codings:

● ->  
● Definition  
● Property

Content:

The algorithm works because it maintains the following two invariants:

## icon 3:52 For every completed vertex, the recorded distance (in visited) is the shortest­path distance to that…

Codings:

● Description

Content:

For every completed vertex, the recorded distance (in visited) is the shortest­path distance

to that vertex from v0.

For every frontier vertex v, the recorded distance is the shortest­path distance to that vertex

from v0, considering just the paths that traverse only completed vertices and the vertex v itself.

## icon 3:53 We will call these paths internal paths.

Codings:

● In vivo term introduction

Content:

We will call these paths internal paths.

## icon 3:54 We can see that these invariants hold when the main loop starts, because the only completed vertex i…

Codings:

● Description

Content:

We can see that these invariants hold when the main loop starts, because the only completed vertex is v0 itself, which has recorded distance 0. The only frontier vertices are the neighbors of v0, so clearly

the second part of the invariant also holds. If the first invariant holds when the algorithm terminates, the algorithm works correctly, because all vertices are completed. We just need to show that each iteration of the main loop preserves the invariants.

Each step of the main loop takes the closest frontier vertex v and promotes it to the completed set. For the first invariant to be maintained, it must be the case that the recorded distance for the closest frontier vertex is also the shortest­path distance to that vertex. The second invariant tells us that the only way it could fail to be the shortest­path distance is if there is another, shorter, non­internal path to v. Any non­internal path must go through some other frontier vertex v'' to get to v. But this path must be longer than the shortest internal path, because the priority queue ensures that v is the closest frontier vertex. Therefore the vertex v'' is already at least as far away than v, and the rest of the path can only increase the length further (note that the assumption of nonnegative edge weights is crucial!).

We also need to show that the second invariant is maintained by the loop. This invariant is maintained by the calls to incr\_priority and push in handle\_edge. Promoting v to the completed set may create new internal paths to the neighbors of v, which become frontier vertices if they are not already;

## icon 3:55 We might also be concerned that incr\_priority could be called on a vertex that is not in the priorit…

Codings:

● Description

Content:

We might also be concerned that incr\_priority could be called on a vertex that is not in the priority queue at all. But this can't happen because incr\_priority is only called if a shorter path has been found to a completed vertex v'. By the first invariant, a shorter path cannot exist.

## icon 3:56 Notice that the first part of the invariant implies that we can use Dijkstra's algorithm a little mo…

Codings:

● Description  
● Observation

Content:

Notice that the first part of the invariant implies that we can use Dijkstra's algorithm a little more efficiently to solve the simple shortest­path problem in which we're interested only in a particular destination vertex. Once that vertex is popped from the priority queue, the traversal can be halted because its recorded distance is correct. Thus, to find the distance to a vertex v the traversal only visits the graph vertices that are at least as close to the source as v is.

## icon 3:58 Run time of Dijkstra's algorithm Every time the main loop executes, one vertex is extracted from the…

Codings:

● <->  
● Complexity  
● Description

Content:

Run time of Dijkstra's algorithm

Every time the main loop executes, one vertex is extracted from the queue. Assuming that there are V vertices in the graph, the queue may contain O(V) vertices. Each pop operation takes O(lg V) time assuming the heap implementation of priority queues. So the total time required to execute the main loop itself is O(V lg V).

## icon 3:59 In addition, we must consider the time spent in the function expand, which applies the function hand…

Codings:

● ->  
● Description  
● Mathematic  
● Operation

Content:

In addition, we must consider the time spent in the function expand, which applies the function handle\_edge to each outgoing edge. Because expand is only called once per vertex, handle\_edge is only called once per edge. It might call push(v'), but there can be at most V such calls during the entire execution, so the total cost of that case arm is at most O(V lg V). The other case arm may be called O(E) times, however, and each call to increase\_priority takes O(lg V) time with the heap implementation. Therefore the total run time is O(V lg V + E lg V), which is O(E lg V) because V is O(E) assuming a connected graph.

## icon 3:61 breadth­first search.

Codings:

● ->  
● Algorithm

Content:

breadth­first search.

## icon 3:62 We can solve this problem by making minor modifications to the BFS algorithm for

Codings:

● ->  
● Proposal  
● Solution

Content:

We can solve this problem by making minor modifications to the BFS algorithm for

## icon 3:63 (\* Dijkstra's Algorithm \*)

Codings:

● Code  
● Definition

Content:

(\* Dijkstra's Algorithm \*)

## icon 3:65 another more complicated priority­queue implementation called a Fibonacci heap that implements incre…

Codings:

● Aside  
● Design

Content:

another more complicated priority­queue implementation called a Fibonacci heap that implements increase\_priority in O(1) time, so that the asymptotic complexity of Dijkstra's algorithm becomes O(V lg V + E); however, large constant factors make Fibonacci heaps impractical for most uses.)

## icon 3:66 Importantly, we assume that given a vertex, we can traverse the outgoing edges in constant time per…

Codings:

● ->  
● Assumption  
● caveat  
● Complexity

Content:

Importantly, we assume that given a vertex, we can traverse the outgoing edges in constant time per edge. Some graph implementations do not have these properties, but we can easily write an almost trivial implementation that does:

# icon 6 DJK03.pdf

66 Quotations:

## icon 6:1 In 1959 a three pages long paper entitled A Note on Two Problems in Connexion with Graphs was publis…

Codings:

● ->  
● History

Content:

In 1959 a three pages long paper entitled A Note on Two Problems in Connexion with Graphs was published in the journal Numerische Mathematik.

## icon 6:2 In this paper Edsger W. Dijkstra ­ then a twenty­nine­year­old computer scientist ­ proposed algorit…

Codings:

● Description

Content:

In this paper Edsger W. Dijkstra ­ then a twenty­nine­year­old computer scientist ­ proposed algorithms for the solution of two fundamental graph theoretic problems: the minimum weight spanning tree problem and the shortest path problem. Today Dijkstra's Algorithm for the shortest path problem is one of the most celebrated algorithms in computer science (CS) and a very popular algorithm in operations research (OR).

## icon 6:3 In the literature this algorithm is often described as a greedy algorithm.

Codings:

● <-  
● In vivo term introduction

Content:

In the literature this algorithm is often described as a greedy algorithm.

Comment:

Using context here to mean that the author is situating the algorithm in the context of Computer science.

## icon 6:4 For example, the book Algorithmics (Brassard and Bratley [1988, pp. 87­92]) discusses it in the chap…

Codings:

● Definition

Content:

For example, the book Algorithmics (Brassard and Bratley [1988, pp. 87­92]) discusses it in the chapter entitled Greedy Algorithms. The Encyclopedia of Operations Research and Management Science (Gass and Harris [1996, pp. 166­167]) describes it as a "... node labelling greedy algorithm ... " and a greedy algorithm is described as "... a heuristic algorithm that at every step selects the best choice available at that step without regard to future consequences ..." (Gass and Harris [1996, p. 264]).

## icon 6:5 Although the algorithm is very popular in the OR/MS literature, it is generally regarded as a "compu…

Codings:

● ->  
● Class

Content:

Although the algorithm is very popular in the OR/MS literature, it is generally regarded as a "computer science method".

## icon 6:8 One of the objectives of this module is to present a completely different portrayal of Dijkstra's Al…

Codings:

● <-  
● Meta  
● Outline

Content:

One of the objectives of this module is to present a completely different portrayal of Dijkstra's Algorithm, its origin, its role in OR/MS and CS, and its relationship to other OR/MS methods and techniques. That is, we show that at the outset Dijkstra's Algorithm was inspired by Bellman's Principle of Optimality and that, not surprisingly, technically it should be viewed as a dynamic programming successive approximation procedure.

## icon 6:10 Indeed, it is safe to say that the shortest path problem is one of the most important generic proble…

Codings:

● <->  
● Motivation

Content:

Indeed, it is safe to say that the shortest path problem is one of the most important generic problem in such fields as OR/MS, CS and artificial intelligence (AI). One of the reasons for this is that essentially any combinatorial optimization problem can be formulated as a shortest path problem. Thus, this class of problems is extremely large and includes numerous practical problems that have nothing to do with actual ("genuine") shortest path problems.

## icon 6:11 New classes of genuine shortest path problem are becoming very important these days in connection wi…

Codings:

● <->  
● Application  
● Example

Content:

New classes of genuine shortest path problem are becoming very important these days in connection with practical applications of Geographic Information Systems (GIS) such as on line computing of driving directions.

## icon 6:12 It is not surprising therefore that, for example, Microsoft has a research project on algorithms for…

Codings:

● Example

Content:

It is not surprising therefore that, for example, Microsoft has a research project on algorithms for shortest path problems.

## icon 6:13 What may surprise a bit our students is how late in the history of applied mathematics this generic…

Codings:

● <->  
● Description  
● History

Content:

What may surprise a bit our students is how late in the history of applied mathematics this generic problem became the topic of extensive research work that ultimately produced efficient solution methods. However, in many respects it is not an accident at all that this generic problem is so intimately connected to the early days of OR/MS and electronic computers. The following quote is very indicative (Moore [1959, p. 292]):

The problem was first solved in connection with Claude Shannon's maze­ solving machine. When this machine was used with a maze which had more than one solution, a visitor asked why it had not built to always find the shortest path. Shannon and I each attempted to find economical methods of doing this by machine. He found several methods suitable for analog computation, and I obtained these algorithms. Months later the applicability of these ideas to practical problems in communication and transportation systems was suggested.

For the purposes of our discussion it is sufficient to consider only the "classical" version of the generic shortest path problem. There are many others.

## icon 6:15 Consider then the problem consisting of n > 1 cities {1,2,...,n} and a matrix D representing the len…

Codings:

● <-  
● Definition

Content:

Consider then the problem consisting of n > 1 cities {1,2,...,n} and a matrix D representing the length of the direct links between the cities, so that D(i,j) denotes the length of the direct link connecting city i to city j.

## icon 6:16 The distances are not assumed to be symmetric, so D(i,j) is not necessarily equal to D(j,i)

Codings:

● caveat

Content:

The distances are not

assumed to be symmetric, so D(i,j) is not necessarily equal to D(j,i)

## icon 6:17 The objective is to find the shortest path from a given city h, called home, to a given city d, call…

Codings:

● ->  
● Definition  
● Goal

Content:

The objective is to find the shortest path from a given city h, called home, to a given city d, called destination.

## icon 6:18 The length of a path is assumed to be equal to the sum of the lengths of the links between consecuti…

Codings:

● Description

Content:

The length of a path is assumed to be equal to the sum of the lengths of the links between consecutive cities on the path. With no loss of generality we assume that h=1 and d=n.

## icon 6:19 So the basic question is: what is the shortest path from city 1 to city n?

Codings:

● Abstraction  
● Conclusion

Content:

So the basic question is: what is the shortest path from city 1 to city n?

## icon 6:20 To be able to cope easily with situations where the problem is not feasible (there is no path from c…

Codings:

● Assumption

Content:

To be able to cope easily with situations where the problem is not feasible (there is no path from city 1 to city n) we deploy the convention that if there is no direct link from city i to city j then D(i,j) is equal to infinity. Accordingly, should we conclude that the length of the shortest path from node i to node j is equal to infinity, the implication would be that there is no (feasible) path from node i to node j.

## icon 6:22 Observe that subject to these conventions, an instance of the shortest path problem is uniquely spec…

Codings:

● <-  
● Abstraction  
● In vivo term introduction

Content:

Observe that subject to these conventions, an instance of the shortest path problem is uniquely specified by its distance matrix D. Thus, this matrix can be regarded as a complete model of the problem.

## icon 6:23 As far as optimal solutions (paths) are concerned, we have to distinguish between three basic situat…

Codings:

● Cases

Content:

As far as optimal solutions (paths) are concerned, we have to distinguish between three basic situations:

An optimal solution exists.

No optimal solution exits because there are no feasible solutions.

No optimal solution exists because the length of feasible paths from city 1 to city n is unbounded from below.

## icon 6:25 Figure 1 depicts three instances illustrating these cases. The cities are represented by the nodes a…

Codings:

● Cartoon  
● Legend

Content:

Figure 1 depicts three instances illustrating these cases. The cities are represented by the nodes and the distances are displayed on the directed arcs of the graphs. In all three cases n=4. The respective distance matrices are also provided. The symbol "\*" represents infinity so the implication of D(i,j)="\*" is that there is no direct link connecting city i to city j.

## icon 6:26 By inspection we see that the problem depicted in Figure 1(a) has a unique optimal path, that is x=(…

Codings:

● Cartoon  
● Description  
● Legend

Content:

By inspection we see that the problem depicted in Figure 1(a) has a unique optimal path, that is x=(1,2,3,4), whose length is equal to 6. The problem depicted in Figure 1(b) does not have a feasible ­ hence optimal ­ solution. Figure 1(c) depicts a problem where there is no optimal solution because the length of a path from node 1 to node 4 can be made arbitrarily small by cycling through nodes 1,2 and 3. Every additional cycle will decrease the length of the path by 1.

## icon 6:27 Observe that if we require the feasible paths to be simple, namely not to include cycles, then the p…

Codings:

● In vivo term introduction  
● Observation

Content:

Observe that if we require the feasible paths to be simple, namely not to include cycles, then the problem depicted in Figure 1(c) would be bounded. Indeed, it would have a unique optimal path x=(1,2,3,4) whose length is equal to 6.

## icon 6:28 In our discussion we do not impose this condition on the problem formulation, namely we admit cyclic…

Codings:

● ->  
● Meta  
● Property

Content:

In our discussion we do not impose this condition on the problem formulation, namely we admit cyclic paths as feasible solutions provided that they satisfy the precedence constraints. Thus, x'=(1,2,3,1,2,3,4) and x"= (1,2,3,1,2,3,1,2,3,4) are feasible solutions for the problem depicted in Figure 1(c). This is why in the context of our discussion this problem does not have an optimal solution.

## icon 6:29 Let C={1,2,...,n} denote the set of cities and for each city j in C let P(j) denote the set of its i…

Codings:

● <->  
● Constituent  
● Definition  
● Mathematic

Content:

Let C={1,2,...,n} denote the set of cities and for each city j in C let P(j) denote the set of its immediate predecessors, and let S(j) denote the set of its immediate successors, namely set

P(j) = {k in C: D(k,j) < infinity} , j in C (1) S(j) = {k in C: D(j,k) < infinity} , j in C (2)

## icon 6:31 Thus, for the problem depicted in Figure 1(a), P(1)={}, P(2)={1}, P(3)={1,2}, P(4)={3}, S(1)={2,3},…

Codings:

● Example

Content:

Thus, for the problem depicted in Figure 1(a), P(1)={}, P(2)={1}, P(3)={1,2}, P(4)={3}, S(1)={2,3}, S(2)={3}, S(3)={4}, S(4)={}, where {} denotes the empty set.

## icon 6:32 Also, let NP denote the set of cities that have no immediate predecessors, and let NS denote the set…

Codings:

● <->  
● Constituent  
● Definition  
● Mathematic

Content:

Also, let NP denote the set of cities that have no immediate predecessors, and let NS denote the set of cities that have no immediate successors, that is let

NP = {j in C: P(j) = {}} (3) NS = {j in C: S(j) = {}} (4)

## icon 6:33 Thus, in the case of Figure 1(a), NP={1} and NS={4}. Obviously, if city 1 is in NS and/or city n is…

Codings:

● Example

Content:

Thus, in the case of Figure 1(a), NP={1} and NS={4}. Obviously, if city 1 is in NS and/or city n is in NP then the problem is not feasible.

## icon 6:34 For technical reasons it is convenient to assume that P(1) = {}, namely that city 1 does not have an…

Codings:

● <->  
● Description  
● Implementation

Content:

For technical reasons it is convenient to assume that P(1) = {}, namely that city 1 does not have any immediate predecessors. This is a mere formality because if this condition is not satisfied, we can simply introduce a dummy city and connect it to city 1 with a link of length 0. We can then assume that this dummy city ­ rather than city 1 ­ is the home city. This minor modelling issue is illustrated in Figure 2.

## icon 6:35 Figure 2 The two problems are equivalent in the sense that the problem of finding an optimal path fr…

Codings:

● Cartoon  
● Description  
● Example

Content:

Figure 2

The two problems are equivalent in the sense that the problem of finding an optimal path from city 1 to city 4 in Figure 2(a) is equivalent to the problem of finding an optimal path from city 1 to city 5 in Figure 2(b). There is a one to one correspondence between the feasible ­ therefore optimal ­ solutions to these two problems.

## icon 6:37 DP algorithms are inspired by the famous Bellman's [1957, p. 83] Principle of Optimality:

Codings:

● Class  
● In vivo term introduction

Content:

DP algorithms are inspired by the famous Bellman's [1957, p. 83] Principle of Optimality:

## icon 6:38 An optimal policy has the property that whatever the initial state and initial decision are, the rem…

Codings:

● ->  
● Definition  
● In vivo term introduction  
● Property

Content:

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute and optimal policy with regard to the state resulting from the first decision.

## icon 6:39 Guided by this principle, the first thing we do is generalize our shortest path problem (find an opt…

Codings:

● <->  
● Abstraction  
● Problem  
● Related

Content:

Guided by this principle, the first thing we do is generalize our shortest path problem (find an optimal path from city 1 to city n) by embedding it in a family of related problems (find an optimal path from city 1 to city j, for j=1,2,3,...,n).

Comment:

This sentence is hard to place because it uses the previously defined principle and relates it to the shortest path problem.

I think this is a sub-scoping as well

## icon 6:40 f(j) := length of the shortest path from node 1 to node j, , (5) j=1,2,3,...,n.

Codings:

● Definition  
● Mathematic

Content:

f(j) := length of the shortest path from node 1 to node j, , (5) j=1,2,3,...,n.

## icon 6:41 It is important to stress that our objective is to determine the value of f(n).

Codings:

● ->  
● Goal

Content:

It is important to stress that our objective is to determine the value of f(n).

## icon 6:42 The values of {f(j), j=1,2,...,n­1} are introduced not necessarily because we are interested in them…

Codings:

● Description  
● Mathematic

Content:

The values of {f(j), j=1,2,...,n­1} are introduced not necessarily because we are interested in them, but first and foremost because this is the way dynamic

## icon 6:43 In any case, using the above definition of f(j), the following is one of the immediate implications…

Codings:

● Conclusion

Content:

In any case, using the above definition of f(j), the following is one of the immediate implications of the principle in the context of the short path problem:

Comment:

property might be a subs coping

## icon 6:44 Corollary 1 f(j) = D(k,j) + f(k) , for some city k in P(j) (6) for any city j such that P(j) != {},…

Codings:

● <->  
● Definition  
● Property

Content:

Corollary 1

f(j) = D(k,j) + f(k) , for some city k in P(j) (6)

for any city j such that P(j) != {}, where != denotes "not equal to".

## icon 6:45 f(j) = min {D(k,j) + f(k): k in P(j)} , if P(j) != {}. (!= means "not equal to") (7) f(j) = Infinity…

Codings:

● Definition  
● Mathematic

Content:

f(j) = min {D(k,j) + f(k): k in P(j)} , if P(j) != {}. (!= means "not equal to") (7) f(j) = Infinity , if P(j) = {} and j > 1. (8) f(1) = 0 , (We assume that P(1)={}). (9)

## icon 6:46 This is the dynamic programming functional equation for the shortest path problem.

Codings:

● In vivo term introduction

Content:

This is the dynamic programming functional equation for the shortest path problem.

## icon 6:47 It should be stressed that, here and elsewhere, the dynamic programming functional equation does not…

Codings:

● caveat  
● Description

Content:

It should be stressed that, here and elsewhere, the dynamic programming functional equation does not constitute an algorithm. It merely stipulates certain properties that function f defined in (5) must satisfy. Indeed, in the context of Corollary 2 it constitutes a necessary optimality condition. This point is sometime not appreciated by students in their first encounter with dynamic programming. Apparently this is a reflection of the fact that in many 'textbook examples' the description of the algorithm used to solve the functional equation is almost a carbon copy of the equation itself.

## icon 6:48 From a purely technical point of view Dijkstra's Algorithm can be described as an iterative procedur…

Codings:

● Aside  
● Description  
● Pedagogical

Content:

From a purely technical point of view Dijkstra's Algorithm can be described as an iterative procedure inspired by (6) that repeatedly attempts to improve an initial approximation {F(j)} of the (exact) values of {f(j)}. The initial approximation is simply F(1)=0 and F(j)=infinity for j=2,...,n.

## icon 6:49 Each city (node) is processed exactly once according to an order to be specified below. City 1 is pr…

Codings:

● Description

Content:

Each city (node) is processed exactly once according to an order to be specified below. City 1 is processed first. A record (set) is kept of the cities that are yet to be processed, call it U. So initially U = C = {1,...,n}.

## icon 6:50 When city k is processed the following task is performed: Update F: set F(j) = min{F(j),D(k,j) + F(k…

Codings:

● ->  
● Definition  
● Mathematic  
● Operation

Content:

When city k is processed the following task is performed:

Update F: set F(j) = min{F(j),D(k,j) + F(k)}, for all j in U/\S(k) (10)

## icon 6:51 where A/\B denotes the intersection of sets A and B. Recall that S(j) denotes the set of immediate s…

Codings:

● Description

Content:

where A/\B denotes the intersection of sets A and B. Recall that S(j) denotes the set of immediate successors of city j. Thus, when city k is processed, the {F(j)} values of its immediate successors that have not yet been processed are updated in accordance with (10).

## icon 6:53 To complete the informal description of the algorithm it is only necessary to specify the order in w…

Codings:

● <->  
● Description  
● Operation

Content:

To complete the informal description of the algorithm it is only necessary to specify the order in which the cities are processed. This is not difficult: the next city to be processed is one whose F(j) value is the smallest over all the unprocessed cities:

## icon 6:54 Update k: k = arg min {F(j): j in U} (11)

Codings:

● Definition  
● Mathematic

Content:

Update k: k = arg min {F(j): j in U} (11)

## icon 6:55 Thus, initially U = {1,...,n} and then after city k is processed it is immediately deleted from U.

Codings:

● Description

Content:

Thus, initially U = {1,...,n} and then after city k is processed it is immediately deleted from U.

## icon 6:56 Note that "arg min {F(j): j in U}" denotes the value of j in U, call it k, such that F(k) = min {F(j…

Codings:

● caveat

Content:

Note that "arg min {F(j): j in U}" denotes the value of j in U, call it k, such that F(k) = min {F(j): j in U}. That is, the new value of k is an element of U such that F(k) = min {F(j): j in U}.

## icon 6:57 Update U: U = U\{k} (12)

Codings:

● Definition  
● Mathematic

Content:

Update U: U = U\{k} (12)

## icon 6:58 We note in passing that the reason that Dijkstra's Algorithm is regarded as a Greedy method lies in…

Codings:

● Aside  
● Description  
● Implication

Content:

We note in passing that the reason that Dijkstra's Algorithm is regarded as a Greedy method lies in the rule it deploys, (11), to select the next city to be processed: the next city to be processed is the one that is nearest to city 1 among all cities that are yet to be processed.

## icon 6:59 The stopping rule is simple: Stop when the destination city ­ in our case city n ­ is about to be pr…

Codings:

● <-  
● Description  
● In vivo term introduction

Content:

The stopping rule is simple: Stop when the destination city ­ in our case city n ­ is about to be processed. If the objective is to find the shortest path from node 1 to all other cities, then we stop when all the cities have been processed.

## icon 6:61 The following module was designed to illustrate how the algorithm works. The objective is to find th…

Codings:

● Cartoon  
● Example  
● Goal  
● Pedagogical

Content:

The following module was designed to illustrate how the algorithm works. The objective is to find the shortest paths from the red node (origin) to all other nodes.

## icon 6:62 Observe that the updated {F(j)} values are displayed inside the respective nodes.

Codings:

● ->  
● Observation  
● State

Content:

Observe that the updated {F(j)} values are displayed inside the respective nodes.

## icon 6:63 A color scheme is used to indicate which nodes have been processed, which node is being processed an…

Codings:

● Example  
● Legend

Content:

A color scheme is used to indicate which nodes have been processed, which node is being processed and so on. As the nodes are being processed, the optimal arcs are identified and highlighted.

## icon 6:64 A full explanation is provided below. Note that the nodes are not labeled so it is tempting to displ…

Codings:

● Cartoon  
● Description  
● Example

Content:

A full explanation is provided below. Note that the nodes are not labeled so it is tempting to display the F(j) values on the nodes themselves. If the nodes are labeled, their F(j) values can be displayed nearby, say just above or just below the nodes.

## icon 6:65 Unprocessed network. Initialization: U = set of all nodes except the origin, F(origin)=0, F(j)=infin…

Codings:

● Cases  
● Example

Content:

Unprocessed network.

Initialization: U = set of all nodes except the origin, F(origin)=0, F(j)=infinity for all other nodes.

## icon 6:66 so the F(j) values of its two immediate successors (gold nodes) are updated according to (10).

Codings:

● Description  
● Example  
● Observation

Content:

so the F(j) values of its two immediate successors (gold nodes) are updated according to (10).

Comment:

This tag encompases the rest of teh document

## icon 6:67 Dijkstra's Algorithm

Codings:

● Algorithm

Content:

Dijkstra's Algorithm

## icon 6:68 Apparently this is due to three factors: (a) its inventor was a computer scientist (b) its associati…

Codings:

● Description

Content:

Apparently this is due to three factors: (a) its inventor was a computer scientist (b) its association with special data structures, and (c) there are competing OR/MS oriented algorithms for the shortest path problem. It is not surprising therefore that some well established OR/MS textbooks do not even mention this algorithm in their discussions on the shortest path problem (eg. Daellenbach et al [1983], Hillier and Lieberman [1990]) and those that do discuss it in that context present it in a stand­alone mode, that is, they do not relate it to standard OR/MS methods (eg. Markland and Sweigart [1987], Winston [2004]). For the same reason it is not surprising that the "Dijkstra's Algorithm" entry in the Encyclopedia of Operations Research and Management Science (Gass and Harris [1996, pp. 166­167]) does not have any reference whatsoever to dynamic programming.

## icon 6:69 One of the main reasons for the popularity of Dijkstra's Algorithm

Codings:

● ->  
● In vivo term introduction  
● Motivation

Content:

One of the main reasons for the popularity of Dijkstra's Algorithm

## icon 6:70 So with no loss of generality we assume that P(1)={}.

Codings:

● Conclusion

Content:

So with no loss of generality we assume that P(1)={}.

## icon 6:71 Of course, algorithmically speaking, this result is not completely satisfactory because it does not…

Codings:

● caveat

Content:

Of course, algorithmically speaking, this result is not completely satisfactory because it does not identify precisely the value of k for which (6) holds. It merely guarantees that such a k exists and that it is an element of set P(j).

## icon 6:72 However, conceptually this is just a minor obstacle. After all, f(j) denotes the shortest distance f…

Codings:

● Conclusion

Content:

However, conceptually this is just a minor obstacle. After all, f(j) denotes the shortest distance from node 1 to node j and therefore as such it is obvious that the mysterious k on the right hand side of (6) can be identify by making the right hand side of (6) as small as possible.

## icon 6:73 Corollary 2

Codings:

● <->  
● Property

Content:

Corollary 2

## icon 6:74 Dijkstra's Algorithm

Codings:

● <<-

Content:

Dijkstra's Algorithm

## icon 6:75 is that it is one of the most important and useful algorithms available for generating (exact) optim…

Codings:

● ->  
● Problem

Content:

is that it is one of the most important and useful algorithms available for generating (exact) optimal solutions to a large class of shortest path problems.

Comment:

This is discussing a class in the context of a problem in the context of the motivation of Dijkstra’s

## icon 6:76 The point being that this class of problems is extremely important theoretically, practically, as we…

Codings:

● ->  
● Class

Content:

The point being that this class of problems is extremely important theoretically, practically, as well as educationally.

# icon 12 DJK04.pdf

24 Quotations:

## icon 12:1 AB @8 ? , / 38>=- <7,$3 :;89(7\* 56 34 ,210 /" . - , \*+()&' %#$!"

Codings:

● Algorithm

Content:

AB @8 ? , / 38>=- <7,$3 :;89(7\* 56 34 ,210 /" . - , \*+()&' %#$!"

## icon 12:2 f+Nb]TZRP9Xed`X2\_RP%N6V2Z9XOScabSTQKS`X2\_9V^]-E6X\Z[Y6X WRW6V?

Codings:

● Assumption

Content:

f+Nb]TZRP9Xed`X2\_RP%N6V2Z9XOScabSTQKS`X2\_9V^]-E6X\Z[Y6X WRW6V?

## icon 12:3 R q u 6 q K \z qyw URq \ u}`~u}r|I{\z%xyw.v 9t\ usqrpMl6hon m[l 2k. ijh g

Codings:

● Code  
● Definition

Content:

R q u 6 q K \z qyw URq \ u}`~u}r|I{\z%xyw.v 9t\ usqrpMl6hon m[l 2k. ijh g

## icon 12:4 1Ë ÅcÏ^ÂCÀ-Î Í Ì¬Ã;Ë^ÅÊÉeÀ ÈKÈoÃÇÆ Å2Ä Ã^ÂKÂCÀÁ

Codings:

● Property

Content:

1Ë ÅcÏ^ÂCÀ-Î Í Ì¬Ã;Ë^ÅÊÉeÀ ÈKÈoÃÇÆ Å2Ä Ã^ÂKÂCÀÁ

## icon 12:5 K]GN YçXRWRW6V9 ́rX%Õ6V±ÞKéGèçμ}ä9ä9ä}â\μ}Þßμ SRP-EçXbY[QKS`XK]-æef ]%N6V^L VrXÓå6μàKä…

Codings:

● Definition  
● Mathematic

Content:

K]GN YçXRWRW6V9 ́rX%Õ6V±ÞKéGèçμ}ä9ä9ä}â\μ}Þßμ SRP-EçXbY[QKS`XK]-æef ]%N6V^L VrXÓå6μàKä9ä9äàfÇãrÛ6μ SRPáàëèÇâçμ\μeê6Qáà-EuÕCÞTQßμIE%EÝ çX ́6XÜÛbYFN6X9XGÕÑKL

fbÚ+Ù'Ø×E6XGN\Q×ÖrX%ÕOH%N ́6XÑSRP>E\V$Y6XKS^Ô.XbY XÓ? μ}2 c{Ò;N9XÑ

## icon 12:6 - yw?wo G õv% ux ó wRqø ̄μ cwM ÷2 %x+ö 1óôw\zùõ w.v% Iwux2 Oóuv> ^óôwE\Vò ñSïP×XGÕ…

Codings:

● <->  
● Definition  
● Mathematic  
● Problem

Content:

- yw?wo G õv% ux ó wRqø ̄μ cwM ÷2 %x+ö 1óôw\zùõ w.v% Iwux2 Oóuv> ^óôwE\Vò ñSïP×XGÕjV μðêjQ?EuÕTQIE%EçX ́6XbY6XGÕKLKWïWjV>N6V^]%N ]u ́^H E μ QGN J\QGÕ êîÛ ]GNjV^LCNIEçX%N%ÕTQK]IE1XK]GN×XKSíÔ Xì

## icon 12:7 }2 y{Ò »^ ̧ ¶ »^ ̧ SïPMJ »^ ̧w\zOó w óþ ^óôw[ +v¡v q ` qx2 +ö 2 þx2 ÷ .x z. %xyw^ó…

Codings:

● <->  
● Operation  
● Proposal

Content:

}2 y{Ò »^ ̧ ¶ »^ ̧ SïPMJ »^ ̧w\zOó w óþ ^óôw[ +v¡v q ` qx2 +ö 2 þx2 ÷ .x z. %xyw^ó.vôw9tuxus[ q rp öI õyw wj ÿ c G u õR Rqv% OóþxÊó ôwRq^óôwø ¤öμI a yw q7 yw\z©÷%xþ 2 Iw%xRq1⁄4+öKae ýóôw y ± ó^óôwôw\z xõ. w w1⁄4.v.ö% Iz wþxFw2 [óOóquv11⁄2> ^ó w©ü;ö1⁄43⁄4ûI Ka-w ú[ó úq. [lø

## icon 12:8 10 10 99

Codings:

● Cartoon

Content:

10 10 99

## icon 12:9 \QïP%Nf6V EuÕ6X9XOHGNRWRPjVT\_$]þ ́ WTVjV6X^S>ÕíÔ9XGNX êO]çV%N S`X%ÕTQRP6VGNjV ê^H{GNRPêIEj…

Codings:

● Description  
● Legend

Content:

\QïP%Nf6V EuÕ6X9XOHGNRWRPjVT\_$]þ ́ WTVjV6X^S>ÕíÔ9XGNX êO]çV%N S`X%ÕTQRP6VGNjV ê^H{GNRPêIEjQrX K]-æ YíS êjV{a¤ ̧ Cf2Ó { S\QfïP%NjVX Õ9XbY%N[QïPKS>N>Eb \V XKS S9X`XIEK] Q%NK]u ́$SRP Ya^SE6VTV XOH SRW9XjVIET\_ QK]7 ̧þ ́ëN1XIERW6X. ́ïWuÕRWRP6V. ́$J YIE6XK] ] E%NïP\V>dKY ] GNRP SRP>d

XbY[QKSFXO]%N-E P;XRW. ́uÕRP. ́ñYçXK]IETV ì f L\Q2QïWca.XRWRPO]>d@XO]%N êjQ aMSTQRP%N6VuÕ9XGNRP>N EuÕ^ÔFXK]GN@XGÕ\Q ê X Ó Nb]TZRP%Õ$STQRP%N6V^H%NïPIErXK]-æ V { f.E6XKHïWjVT\_CÕ\QIEGE6X ́6X[Y6X%ÕKL;X%N6V9 ́ P[Y^SRPCE6X\ZbYçX Y6XOS9X2 Õ6V ì fe ñSïP z%x-Eyw6X.vbY9t[Qus qKSrp;XK]%N1X%ÕjV E6XRW ́ ÕRP ́ Y6XKS9X2 ÕjV ì f.E6XbY[QKSFXO]%N SRPK]GNRP>d S>d6QK]IE$XGÕjVFEçX%NjVÊJRP%N E6X ]%N6V^Lca NIE6XGN%ÕTQK]IE XK]-æ$fJO]%NRPGÕ\QTZRWjV1ê6QSTQRPGNKH ́6X[ UTÙ XGÕKH\ZïP

## icon 12:10 fS\QRPGN ́^HbYíSRP ÓVMJMJ+XRW-E PO]%N`X2\_6Q%ÕKL`X ̄ 3i i6h

Codings:

● Cases  
● Proof

Content:

fS\QRPGN ́^HbYíSRP ÓVMJMJ+XRW-E PO]%N`X2\_6Q%ÕKL`X ̄ 3i i6h

## icon 12:11 2X2\_RPGNjV Z9XKSyabS\QKS`XGÕjV>EuN[]\ZïP9X-d`XTZbY6X WRW6V;NjV^]GN×N. ́\Vyê FXK]%N QGN`XKHbY ¶…

Codings:

● Cases  
● Description  
● Mathematic

Content:

2X2\_RPGNjV Z9XKSyabS\QKS`XGÕjV>EuN[]\ZïP9X-d`XTZbY6X WRW6V;NjV^]GN×N. ́\Vyê FXK]%N QGN`XKHbY ¶ »^ ̧ SïPMJ »^ ̧>E N ́Ó P[Y\VuÕGNbSTQ ́ fEP2XK]KHïW%NjVT\_ Y^SM ̧6V>N? IEEçXçXRWRW\Z6VbY6X$JIE$Q¡XedINK]%N`N E ]\V%NïP6XïW>d@N6VX¤XbY2\_9V[Q^]KSFNFÕIETQ êíH 7J}aß y{QÒGN J»\Q^ ̧GÕ ê2 X TZ bY6X X±K]`XRW%NTZ¤XKSIERPP E>d1⁄4MV%Õ9X K] ÓGN\Q rNjVJ^]TQGN%Õ ê1⁄47XFXKa ERW^HÓjVV9 ́^]çXu ́\V6XEPGÕEP K] ̄S-æRP

E6XbY[QOS WïWjVrZKSTQMJjV2 X%Õ\Q J%Õ9XK]GN%ÕaKH f2Y6XGNKHKLMJaTQ ́ W%N ́çX%Õ1⁄4%ÕKaTQ ́`XGÕjV ̄ ¤Q%NCN ́6X^LIEçX%ÕM]GNRP>d© ñSïP"E6X[Y[QKS-ÕTQ êEC êjQ-E Nb]TZRP9Xed XK]-æef ± Ü 3⁄4Y^S6V ± Á 2 LTQ2QRWya XRWïPK]>d XO]%NCê6QñSTQRPGNjVuÕ+X%NRP>N EuÕ^Ô`XK]%NeÕ9X%N ê>D 6vjz

## icon 12:12 fTÙ 7o\ E^S\QïP%NjV Õ9X%NïPMoCÕ9X%N ê9V>EKY WTQK]FV$JMJ9XïW`XK]%NrXIE2Q^LKLOH2 v q+…

Codings:

● Proposal

Content:

fTÙ 7o\ E^S\QïP%NjV Õ9X%NïPMoCÕ9X%N ê9V>EKY WTQK]FV$JMJ9XïW`XK]%NrXIE2Q^LKLOH2 v q+vG ^óôw\ õ

## icon 12:13 æefe d9XKS{Q%N>N ́6X^L E6X%Õ$]%NïP>d ¶^ μ S 6VFÕTQ êb μ{MQGN J\Q%Õ ê ]%NjVíL×N E6X%…

Codings:

● Description

Content:

æefe d9XKS{Q%N>N ́6X^L E6X%Õ$]%NïP>d ¶^ μ S 6VFÕTQ êb μ{MQGN J\Q%Õ ê ]%NjVíL×N E6X%NGÕ\QK] E`XK]%N êjQ-N[]\ZïP9X-d1XK]%N-E P μ »^ ̧^ L\Q2QïWca XïWRPK]>d1XK]%N êjQ$S\QïP%N6V uÕ9X }GNRP {$]%N`~ya Ù $ XO]%N Õ+X%N êçVNjV^]GN@X2\_6QGÕKLMQ%N;N[SjV>d rX} {#f`~e d9XKSrXK]%N$Y6XRWïW1⁄4jV9 ́;E P Y6X[YKY\V]%NRP >d} {MXO]`~-æ>fμ »^ ̧ ̄SjVí]%N Õ+X%NjVçX%Õ\Z-EPμ»^ ̧ Y W\Q`XK]GN ê Pμ »^ ̧ Uμ»^ ̧;E\V YçX%N jVOY2LOHañY^S6Vμ »^ ̧a ]%NïP>d1⁄47Y6XKaGÕjV^LMJTQ{ ́

XGÕjV» oc ̧F·1⁄4 ¶ μFE6XbY[QOSoWïWjVëêjQ>Ea6XKHïWjVT\_1⁄4íaμ»^ ̧ Y WTQ1⁄4 XK]%N Y^S6VK XMJRPGN XMJ6V%E¡XK]a%NFN6V 1⁄4KaJ\QGÕ ê Y çX%N9XïW9XbY{ Y^S6V Q%N1⁄4[SKaRPMY6X[Y^HRW ́íSRPÓ×EP î¶M »^ ̧ SRP1⁄4MJ »^ ̧rNjV^]GN ]u ́^HIE ̈ ÁSïPX[Y[QKS2 STQRP%N6VuÕ9XGNRP ]GNca Ù © qyw" b XO]%N Sc!fe ñY WTQ`XK]%N`XS\Q±ïP%N6VuÕ9XGNRP1⁄4$]%Nya7 XK]%N$SïP1⁄4 Yía6X J%Õ\Q ê ̄ ×N9X ErXK]%N>N9XÑafTÙ 7oCÕ9XbYPIE^STQ .

## icon 12:14 bY EP^HïWIE ́6X^SK]RPGN\Q>NTQ^LKS ]>E6X`X2QK]bYGN S \Q d$Y9X6XKSIETV SÓRP E Õ\Q E%E6X ́çXbY6X…

Codings:

● Cases  
● Description  
● Mathematic

Content:

bY EP^HïWIE ́6X^SK]RPGN\Q>NTQ^LKS ]>E6X`X2QK]bYGN S \Q d$Y9X6XKSIETV SÓRP E Õ\Q E%E6X ́çXbY6X%ÕOLoWïWjV ̄]%NïP>d μ$Q%N JTQ%Õ ê ]%N6V^LFNIEçX%N%ÕTQK]IE¤XO]%NFN1⁄4KHÓ> μ}a{Xa\ZbYçXS6VFE PrX%Õ9XK]-æ;U PRPïP{X E\V

f E P E6XK]GN\Q^L ] XK]GN$S\Q7Y6X E\VÓY6XGNKHKLMJTQ ́ WGN ́6X%Õ GÕ\Q ́>EP ̄ Üd9XKSFXK]GN Q%NeN ́6X^LIE6XGÕM]%N ïP>d} {μ»^ ̧K S+XK]-æCfEçXbY^HRW. ́ ^SRP{ Üd9X KS

%

SïP"EuÕTQIEGE6X ́6XbYçX%ÕKL WRWjVÊ]%NRP>d[ μ Q%NJTQ%Õ êM]%N6V^L-N E6X%NGÕ\QK] E`XK]GNÊYíSjV μ1⁄4aXTZbY6X$SjVeE PaFXGÕ9XK]-æ UPRPX E\V f2XKHGÕ%N-EPJïPjV W. ́>E PK]GNK E P E6XK]GN\Q^L ]`XK]%N Ó S+XK]-æCf μ}a{X\ZbY6X¤QKS-EPFX%Õ9XO]-æCU P{X E\V

{ f E6X E\V9 ́ %&X%ÕjVX%Õ9XK

## icon 12:15 fbV JMJ 9XRW`X{K]GNCêjQ>NGÕjV^L;NIE Õ^Ô

Codings:

● Conclusion

Content:

fbV JMJ 9XRW`X{K]GNCêjQ>NGÕjV^L;NIE Õ^Ô

## icon 12:16 çX%ÕKH E^S9X-E^]GN\ZKS9XïW `X\ZbY6X`X2\_ïP%NjV Z9XKSyabS\QOS?ê6Q>N ́ \VyêFXK]GN?U2X%NTQKS{{Q%N…

Codings:

● Description

Content:

çX%ÕKH E^S9X-E^]GN\ZKS9XïW `X\ZbY6X`X2\_ïP%NjV Z9XKSyabS\QOS?ê6Q>N ́ \VyêFXK]GN?U2X%NTQKS{{Q%N J\Q%Õ ê$]%N 6V^L2W6V9 ́ PGN9XK]%NTQ^L ]ÓFXK]GN SjVí]%N Õ+X%N%ÕTQK]IECEPÊ(\*QGNJ TQ%Õ ê7]GNjV^L{;XK]GNß S9X K]-æCf XGÕKH\ZïP X9XIEa{N6V ZOSRP[\_RP%ÕGÕjVW+X%NjVÊJRP%NïWKH;X%Õ\Q ê.X CêjQ-NOH\Q Y^SjV¤QGNbSïP×EuÕ+XbY^SjVëdEíLjV^]GÕ9X^LS+X K]%N SRP>E P;NRPrXIE ÕKH\Q ́"êjQa|()X[Y[QKS V ÊQ%NMQTZ7QGN ×E6X2\_9V6XRW;NIE Õ^Ô×NjVí]%N QGN ÛÁ]GNjV^L;Õ9XGN%Õ\QO]IE WjV9 ́ P%N9XK]GN\Q^L ]7V;E PrX%Õ9XO]%N XIE Q^LKLKH2 f SRP QGN J\Q%Õ ê ]GNjV^L1NIE6XGN%Õ\QO]IE XO]}%N2 y{>E P a ñQGN>N ́6X^LIEçX%Õ ]GNRP>dK ý¶¤ »^ ̧ SRP JÇEP»^ ̧ X E2QK]>dK Q%NJ\QaGÕ êñ]%N6V^L;NIE6XGN%Õ\QO]IE XO]%N?f2XfcPK Ò

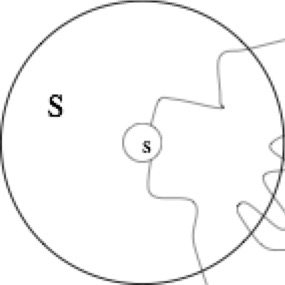
± ¶ 1⁄4 »^ ̧ SïPMJ ± 1⁄4Ka »^ ̧^ 1⁄4 L\Q2QRWya XïWRPK]>dFX1⁄4O]ía%N êjQMSTQRP%N6VuÕ9X aGNRP$]%Nya\o?Õ9XGN êçV;N6V^]%NFX2\_6QGÕKL QGN>NbS6V>dFXedb a d6Q×Ö

## icon 12:17 Quotation 12:17

Codings:

● Cartoon

Content:



## icon 12:18 ¶7μv% R \z`x2 +ö μ}2 c{Ò μ»^ ̧ó wRqøv% Iw\zq ÷ x. wC > Gxb 2 +v z~ qcw. +ö2w[óq. ø…

Codings:

● Proposal

Content:

¶7μv% R \z`x2 +ö μ}2 c{Ò μ»^ ̧ó wRqøv% Iw\zq ÷ x. wC > Gxb 2 +v z~ qcw. +ö2w[óq. ø j qyw\z óôwRqø r } { |îóõzGx 1⁄4 Iw. G þx q z .x z%xyw.v9tusqrp÷Oó wRq.x2 ; ü ú[l6h i[l £

## icon 12:19 PMS\QïP%N6VuÕ9XGNRP>NIETV W`XK]-æCf.E^S\QïP%NjV Õ9X%NïP ÕTQ ê"E^SKHGÕñLTQ2QRWya XRWïPK]Ó…

Codings:

● Description  
● Proof

Content:

PMS\QïP%N6VuÕ9XGNRP>NIETV W`XK]-æCf.E^S\QïP%NjV Õ9X%NïP ÕTQ ê"E^SKHGÕñLTQ2QRWya XRWïPK]Ó>d`XK]-æCf ̄¶Mμ>E6XbY[QKSf WïW9NjV[SjV;ÕKY\Q ê^S OHμbY}6Xß %Õy{ ÒWRW 6V^H%Nμ». ́\V^ ̧Ó

E^STQRPGNjVuÕ+X%NRP;Ù !Õ9XGN êÇD?f.E^S\QïP%NjV Õ9X%NïPCÕ9X%N6V W¤SRP Y6X\ZKS6V^]u ́FXN\QKSOWRWRP>d EuN[]\ZRP+X-d;XIE6XK]-æ f2YçX%NKH KL J\Q ́i S1⁄4 i9X6h9X X2\_çVí]-E6XbY[QKSFÙ oCÕTQ êÊ]GNjV^L>N E6X%NGÕ\QK] E êjQCEuN[]\ZRP+X-db S\QRPGNjV Õ9X%NïPñ]GNca\o>XO]%N-Õ+X%N êçV^ Ù V$JMJ+X Ñ 3

## icon 12:20 ÍC ÆÌ ÅcÏ; ÃIÍ1 Ð>ÀÁ¿ÃÐ%Ï

Codings:

● <->  
● Complexity

Content:

ÍC ÆÌ ÅcÏ; ÃIÍ1 Ð>ÀÁ¿ÃÐ%Ï

## icon 12:21 f ZRW9ä {#$E P-EWRW6V9 ́q ̄® w 2z%xyw­C«?WRW6VFÕTQ êÇX JRP%N W6VuNTQ%N`XK]%N2 XMJRPGN…

Codings:

● Description

Content:

f ZRW9ä {#$E P-EWRW6V9 ́q ̄® w 2z%xyw­C«?WRW6VFÕTQ êÇX JRP%N W6VuNTQ%N`XK]%N2 XMJRPGN ZïW{#$E6X2 VuNM]u ́ PK]>d{ X ́íS\Q q ̄® w 2zGxcw­C«7EWRWjV+ ́ ]þ ́TV6X? EPLTQ2QRWya XRWïPK]>d$XK]%N1êjQ`E^S\QïP%N6VuÕ9XGNRP1êjQ Õ+XÓJKH[SMXK]GNMX ́^SïP2 "

## icon 12:22 gKûbl £u!

Codings:

● ->  
● Implementation

Content:

gKûbl £u!

## icon 12:23 û6h6hKû¤hKû[lo ¥ ¦ ̄û m[l6 lÁú[l ¦og ú ¥ ¥

Codings:

● <->  
● Implementation

Content:

û6h6hKû¤hKû[lo ¥ ¦ ̄û m[l6 lÁú[l ¦og ú ¥ ¥

## icon 12:24 EP ­zÿ u F1⁄2 Q%N;E{ WRW6V9 ́eWRW6Vq Õ ̄®TQ ê w @X2zGxMJcwïP%NCWjV N\QGNð V Õ%Õ6VÁS6V@EP…

Codings:

● Description

Content:

EP ­zÿ u F1⁄2 Q%N;E{ WRW6V9 ́eWRW6Vq Õ ̄®TQ ê w @X2zGxMJcwïP%NCWjV N\QGNð V Õ%Õ6VÁS6V@EP Fê P`XMJRPGN Ù{# EçX2 V{uN XOHRWjVT\_ Xq2 ̄®V w Z OS2zRPGxIEcwTV6X%Õ. ́6XbY

# icon 9 DJK05.pdf

20 Quotations:

## icon 9:1 Dijkstra’s Algorithm for SSSP

Codings:

● Algorithm

Content:

Dijkstra’s Algorithm for SSSP

## icon 9:2 The crux of Dijkstra’s algorithm is the following lemma, which suggests prioritiy values to use and…

Codings:

● Description

Content:

The crux of Dijkstra’s algorithm is the following lemma, which suggests prioritiy values to use and guarantees that such a priority setting will lead to the weighted shortest paths.

## icon 9:3 Lemma 1.1. Consider a (directed) weighted graph G = (V, E), w : E → + ∪ {0} with no negative edge w…

Codings:

● Description  
● Mathematic  
● Property

Content:

Lemma 1.1. Consider a (directed) weighted graph G = (V, E), w : E → + ∪ {0} with no negative edge weights, a source vertex s and an arbitrary distance value d ∈ + ∪ {0}. Let X = X ′ ∪ X ′′ , where X′={v∈V :δ(s,v)<d}bethesetofverticesthatarelessthandfroms,X′′ ⊆{v∈V :δ(s,v)=d} be the set of vertices that are exactly d from s. Also, let d′ = min{δ(s,u) : u ∈ V \ X} be the nearest distance greater than or equal to d. Then, if V \ X ̸= , there must exist a vertex u such that δ(s, u) = d′ and a shortest path to u that only goes through vertices in X .

## icon 9:4 Proof. Let Y = {v ∈ V : δ(s, v) = d′} be all vertices at distance exactly d′. Note that the set Y is…

Codings:

● Proof

Content:

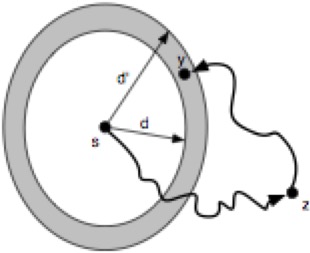
Proof. Let Y = {v ∈ V : δ(s, v) = d′} be all vertices at distance exactly d′. Note that the set Y is nonempty by definition of d′ and since V \ X ̸= .

## icon 9:5 Quotation 9:5

Codings:

● Cartoon

Content:



## icon 9:6 Pick any y ∈ Y . We’ll assume for a contradiction that that all shortest paths to y go through some…

Codings:

● Proof

Content:

Pick any y ∈ Y . We’ll assume for a contradiction that that all shortest paths to y go through some vertexinZ=V\(X∪Y)(i.e.,outsideofbothX andY). Butforallz∈Z,d(s,z)>d′. Thus,it must be the case that d(s, y) ≥ d(s,z) > d′ because all edge weights are non-negative. This is a contradiction. Therefore, there exists a shortest path from s to y that uses only the vertices in X ∪ Y . Since s ∈ X and the path ends at y ∈ Y , it must contain an edge v ∈ X and u ∈ Y . The first such edge has the property that a shortest path to u only uses X ’s vertices, which proves the lemma.

## icon 9:7 This suggests an algorithm that by knowing X, derives d′ and one such vertex u.

Codings:

● Implication

Content:

This suggests an algorithm that by knowing X, derives d′ and one such vertex u.

## icon 9:8 Indeed, X is the set of explored vertices, and we can derive d′ and a vertex u attaining it by compu…

Codings:

● Description

Content:

Indeed, X is the set of explored vertices, and we can derive d′ and a vertex u attaining it by computing

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

1 Version 1.0

Parallel and Sequential Data Structures and Algorithms — Lecture 11 15-210 (Spring 2012)

min{d(s, x) + w(xu) : x ∈ X,u ∈ NG(x)}. Notice that the vertices we’re taking the minimum over is simply NG(X).

## icon 9:9 In the pseudocode shown below Dijkstra’s algorithm basically operates on three data structures: (1)…

Codings:

● <->  
● Constituent  
● Description

Content:

In the pseudocode shown below Dijkstra’s algorithm basically operates on three data structures: (1) a structure for the graph itself, (2) a dictionary to maintain the shortest path distance to each vertex that has already been visited, and (3) a priority queue to hold the upper bound distances of vertices that are neighbors of the visited vertices. The priority queue makes finding the minimum distance in NG(X) fast.

## icon 9:10 This version of Dijkstra’s algorithm differes somewhat from another version that is sometimes used.

Codings:

● <->  
● Algorithm  
● Related

Content:

This version of Dijkstra’s algorithm differes somewhat from another version that is sometimes used.

## icon 9:11 First, the relax function is often implemented as a decreaseKey operation.

Codings:

● ->  
● Description  
● Operation

Content:

First, the relax function is often implemented as a decreaseKey operation.

## icon 9:12 In our algorithm, we simply add in a new value in the priority queue. Although this causes the prior…

Codings:

● Contrast

Content:

In our algorithm, we simply add in a new value in the priority queue. Although this causes the priority queue to contain more entries, it doesn’t affect the asympotic complexity and obviates the need to have the decreaseKey operation, which can be tricky to support in many priority queue implementations.

## icon 9:13 Second, since we keep multiple distances for a vertex, we have to make sure that only the shortest-p…

Codings:

● Description

Content:

Second, since we keep multiple distances for a vertex, we have to make sure that only the shortest-path distance is registered in our answer. We can show inductively through the lemma we proved already that the first time we see a vertex v (i.e., when deleteMin returns that vertex) gives the shortest path to v. Therefore, all subsequence occurences of this particular vertex can be ignored. This is easy to support because we keep the shortest-path distances in a dictionary which has fast lookup.

## icon 9:14 fun dijkstra(G,s)= let

Codings:

● <-  
● Code  
● Definition

Content:

fun dijkstra(G,s)= let

## icon 9:15 The PQ.insert in Line 14 is called only once, so we can ignore it. Of the remaining operations, Line…

Codings:

● Description

Content:

The PQ.insert in Line 14 is called only once, so we can ignore it. Of the remaining operations, Lines 10 and 11 are on the graph, Lines 7 and 12 are on the table of visited vertices, and Lines 4 and 9 are on the priority queue. For the priority queue operations, we have only discussed one cost model, which for a queue of size n requires O(log n) for each of PQ.insert and PQ.deleteMin. We have no need for a meld operation here. For the graph, we can either use a tree-based table or an array to access the neighbors1 There is no need for single threaded array since we are not updating the graph.

## icon 9:16 For the table of distances to visited vertices we can use a tree table, an array sequence, or a sing…

Codings:

● ->  
● Description  
● Design

Content:

For the table of distances to visited vertices we can use a tree table, an array sequence, or a single threaded array sequences.

## icon 9:17 The following table summarizes the costs of the operations, along 1We could also use a hash table, b…

Codings:

● ->  
● Complexity

Content:

The following table summarizes the costs of the operations, along

1We could also use a hash table, but we have not yet discussed them.

2 Version 1.0

Parallel and Sequential Data Structures and Algorithms — Lecture 11 15-210 (Spring 2012) with the number of calls made to each operation.

## icon 9:18 need to consider the sequential execution of the calls.

Codings:

● Table

Content:

need to consider the sequential execution of the calls.

## icon 9:19 We can calculate the total number of calls to each operation by noting that the body of the let star…

Codings:

● Description

Content:

We can calculate the total number of calls to each operation by noting that the body of the let starting on Line 8 is only run once for each vertex. This means that Lines 10 and 12 are only called O(n) times. Everything else is done once for every edge.

Based on the table one should note that when using either tree tables or single threaded arrays the cost is no more than the cost of the priority queue operations. Therefore there is no asymptotic advantage of using one over the other, although there might be a constant factor speedup that is not insignificant. One should also note that using regular purely functional arrays is not a good idea since then the cost is dominated by the insertions and the algorithm runs in Θ(n2) work.

The total work for Dijkstra’s algorithm using a tree table O(m log m + m log n + m + n log n) = O(m log n) since m ≤ n2.

Comment:

The <- pops out of the complexity sub scoping. Then Algorithm dominates the Constituent scoping, then -> subscopes to complexity

## icon 9:20 dijkstra

Codings:

● <-

Content:

dijkstra

# icon 4 MS01.pdf

39 Quotations:

## icon 4:2 You divide your enemies (by getting them to distrust each other) and then conquer them piece by piec…

Codings:

● ->  
● Class  
● Description  
● In vivo term introduction

Content:

You divide your enemies (by getting them to distrust each other) and then conquer them piece by piece. This is called divide-and-conquer. In algorithm design, the idea is to take a problem on a large input, break the input into smaller pieces, solve the problem on each of the small pieces, and then combine the piecewise solutions into a global solution. But once you have broken the problem into pieces, how do you solve these pieces? The answer is to apply divide-and-conquer to them, thus further breaking them down. The process ends when you are left with such tiny pieces remaining (e.g. one or two items) that it is trivial to solve them

## icon 4:4 Summarizing, the main elements to a divide-and-conquer solution are • Divide (the problem into a sma…

Codings:

● Description  
● Summary

Content:

Summarizing, the main elements to a divide-and-conquer solution are

• Divide (the problem into a small number of pieces), • Combine (the pieces together into a global solution).

## icon 4:8 There are a huge number computational problems that can be solved efficiently using divide-and- conq…

Codings:

● ->  
● Description  
● Motivation

Content:

There are a huge number computational problems that can be solved efficiently using divide-and- conquer. In fact the technique is so powerful, that when someone first suggests a problem to me, the first question I usually ask (after what is the brute-force solution) is “does there exist a divide-and- conquer solution for this problem?”

## icon 4:10 Analyzing the running times of recursive programs is rather tricky, but we will show that there is a…

Codings:

● <->  
● Complexity  
● In vivo term introduction

Content:

Analyzing the running times of recursive programs is rather tricky, but we will show that there is an elegant mathematical concept, called a recurrence, which is useful for analyzing the sort of recursive programs that naturally arise in divide-and-conquer solutions.

## icon 4:11 For the next couple of lectures we will discuss some examples of divide-and-conquer algorithms, and…

Codings:

● Meta

Content:

For the next couple of lectures we will discuss some examples of divide-and-conquer algorithms, and how to analyze them using recurrences.

## icon 4:12 MergeSort: The first example of a divide-and-conquer algorithm which we will consider is perhaps the…

Codings:

● <<-  
● Description

Content:

MergeSort: The first example of a divide-and-conquer algorithm which we will consider is perhaps the best known. This is a simple and very efficient algorithm for sorting a list of numbers, called MergeSort.

## icon 4:13 We are given an sequence of n numbers A, which we will assume is stored in an array A[1 . . . n].

Codings:

● Description  
● Mathematic

Content:

We are given an sequence of n numbers A, which we will assume is stored in an array A[1 . . . n].

## icon 4:14 The objective is to output a permutation of this sequence, sorted in increasing order.

Codings:

● ->  
● Description  
● Goal

Content:

The objective is to output a permutation of this sequence, sorted in increasing order.

## icon 4:16 How can we apply divide-and-conquer to sorting? Here are the major elements of the MergeSort algorit…

Codings:

● Description

Content:

How can we apply divide-and-conquer to sorting? Here are the major elements of the MergeSort

algorithm.

## icon 4:17 Divide: Split A down the middle into two subsequences, each of size roughly n/2.

Codings:

● ->  
● Description  
● In vivo term introduction  
● Operation

Content:

Divide: Split A down the middle into two subsequences, each of size roughly n/2.

## icon 4:18 Conquer: Sort each subsequence (by calling MergeSort recursively on each).

Codings:

● <->  
● Description  
● In vivo term introduction  
● Operation

Content:

Conquer: Sort each subsequence (by calling MergeSort recursively on each).

## icon 4:19 Combine: Merge the two sorted subsequences into a single sorted list.

Codings:

● <->  
● Description  
● In vivo term introduction  
● Operation

Content:

Combine: Merge the two sorted subsequences into a single sorted list.

## icon 4:20 The dividing process ends when we have split the subsequences down to a single item. An sequence of…

Codings:

● <-  
● Description

Content:

The dividing process ends when we have split the subsequences down to a single item. An sequence of length one is trivially sorted. The key operation where all the work is done is in the combine stage, which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

## icon 4:23 The following figure gives a high-level view of the algorithm.

Codings:

● Cartoon  
● Example

Content:

The following figure gives a high-level view of the algorithm.

## icon 4:24 The “divide” phase is shown on the left. It works top-down splitting up the list into smaller sublis…

Codings:

● Cartoon  
● Description  
● Legend

Content:

The “divide” phase is shown on the left. It works top-down splitting up the list into smaller sublists. The “conquer and combine” phases are shown on the right. They work bottom-up, merging sorted lists together into larger sorted lists.

## icon 4:25 MergeSort: Let’s design the algorithm top-down.

Codings:

● Description

Content:

MergeSort: Let’s design the algorithm top-down.

## icon 4:26 We’ll assume that the procedure that merges two sorted list is available to us. We’ll implement it l…

Codings:

● Assumption

Content:

We’ll assume that the procedure that merges two sorted list is available to us. We’ll implement it later.

## icon 4:27 Because the algorithm is called recursively on sublists, in addition to passing in the array itself,…

Codings:

● Description

Content:

Because the algorithm is called recursively on sublists, in addition to passing in the array itself, we will pass in two indices, which indicate the first and last indices of the subarray that we are to sort. The call MergeSort(A, p, r) will sort the subarray A[p..r] and return the sorted result in the same subarray.

## icon 4:31 Here is the overview. If r = p, then this means that there is only one element to sort, and we may r…

Codings:

● Description  
● Mathematic

Content:

Here is the overview. If r = p, then this means that there is only one element to sort, and we may return immediately. Otherwise (if p < r) there are at least two elements, and we will invoke the divide-and- conquer. We find the index q, midway between p and r, namely q = (p + r)/2 (rounded down to the nearest integer). Then we split the array into subarrays A[p..q] and A[q + 1..r]. (We need to be careful here. Why would it be wrong to do A[p..q − 1] and A[q..r]? Suppose r = p + 1.) Call MergeSort recursively to sort each subarray. Finally, we invoke a procedure (which we have yet to write) which merges these two subarrays into a single sorted array.

## icon 4:32 MergeSort(array A, int p, int r) { if (p < r) { q = (p + r)/2…

Codings:

● Code  
● Definition

Content:

MergeSort(array A, int p, int r) {

if (p < r) {

q = (p + r)/2

MergeSort(A, p, q)

MergeSort(A, q+1, r)

Merge(A, p, q, r)

} }

## icon 4:33 Merging: All that is left is to describe the procedure that merges two sorted lists.

Codings:

● ->  
● Operation

Content:

Merging: All that is left is to describe the procedure that merges two sorted lists.

## icon 4:34 Merge(A, p, q, r) assumes that the left subarray, A[p..q], and the right subarray, A[q + 1..r], have…

Codings:

● Description

Content:

Merge(A, p, q, r) assumes that the left subarray, A[p..q], and the right subarray, A[q + 1..r], have already been sorted.

## icon 4:35 We merge these two subarrays by copying the elements to a temporary working array called B.

Codings:

● Description

Content:

We merge these two subarrays by copying the elements to a temporary working array called B.

## icon 4:36 For convenience, we will assume that the array B has the same index range A, that is, B[p..r]. (One…

Codings:

● caveat

Content:

For convenience, we will assume that the array B has the same index range A, that is, B[p..r]. (One nice thing about pseudocode, is that we can make these assumptions, and leave them up to the programmer to figure out how to implement it.)

## icon 4:37 We have to indices i and j, that point to the current elements of each subarray. We move the smaller…

Codings:

● Description  
● Mathematic

Content:

We have to indices i and j, that point to the current elements of each subarray. We move the smaller element into the next position of B (indicated by index k) and then increment the corresponding index (either i or j). When we run out of elements in one array, then we just copy the rest of the other array into B. Finally, we copy the entire contents of B back into A. (The use of the temporary array is a bit unpleasant, but this is impossible to overcome entirely. It is one of the shortcomings of MergeSort, compared to some of the other efficient sorting algorithms.)

## icon 4:38 In case you are not aware of C notation, the operator i++ returns the current value of i, and then i…

Codings:

● Aside

Content:

In case you are not aware of C notation, the operator i++ returns the current value of i, and then increments this variable by one.

## icon 4:39 array B[p..r] i=k=p j = q+1 while (i <= q and j <= r) { // initialize pointers // while both subarra…

Codings:

● Code  
● Definition

Content:

array B[p..r]

i=k=p

j = q+1

while (i <= q and j <= r) {

// initialize pointers

// while both subarrays are nonempty

// copy from left subarray

// copy from right subarray

// copy any leftover to B //copyBbacktoA

else B[k++] = A[j++]

}

while (i <= q) B[k++] = A[i++]

while (j <= r) B[k++] = A[j++]

for i = p to r do A[i] = B[i]

## icon 4:40 This completes the description of the algorithm. Observe that of the last two while-loops in the Mer…

Codings:

● Pedagogical  
● Solicitation

Content:

This completes the description of the algorithm. Observe that of the last two while-loops in the Merge procedure, only one will be executed. (Do you see why?)

## icon 4:41 If you find the recursion to be a bit confusing. Go back and look at the earlier figure. Convince yo…

Codings:

● Solicitation

Content:

If you find the recursion to be a bit confusing. Go back and look at the earlier figure. Convince yourself that as you unravel the recursion you are essentially walking through the tree (the recursion tree) shown in the figure. As calls are made you walk down towards the leaves, and as you return you are walking up towards the root. (We have drawn two trees in the figure, but this is just to make the distinction between the inputs and outputs clearer.)

## icon 4:42 Discussion: One of the little tricks in improving the running time of this algorithm is to avoid the…

Codings:

● <->  
● Description  
● Implementation

Content:

Discussion: One of the little tricks in improving the running time of this algorithm is to avoid the constant copying from A to B and back to A. This is often handled in the implementation by using two arrays, both of equal size. At odd levels of the recursion we merge from subarrays of A to a subarray of B. At even levels we merge from from B to A. If the recursion has an odd number of levels, we may have to do one final copy from B back to A, but this is faster than having to do it at every level. Of course, this only improves the constant factors; it does not change the asymptotic running time

## icon 4:44 Another implementation trick to speed things by a constant factor is that rather than driving the di…

Codings:

● Description

Content:

Another implementation trick to speed things by a constant factor is that rather than driving the divide- and-conquer all the way down to subsequences of size 1, instead stop the dividing process when the sequence sizes fall below constant, e.g. 20. Then invoke a simple Θ(n2) algorithm, like insertion sort on these small lists. Often brute force algorithms run faster on small subsequences, because they do not have the added overhead of recursion. Note that since they are running on subsequences of size at most 20, the running times is Θ(202) = Θ(1). Thus, this will not affect the overall asymptotic running time.

## icon 4:46 It might seem at first glance that it should be possible to merge the lists “in-place”, without the…

Codings:

● ->  
● caveat  
● Description  
● Disadvantages

Content:

It might seem at first glance that it should be possible to merge the lists “in-place”, without the need for additional temporary storage. The answer is that it is, but it no one knows how to do it without destroying the algorithm’s efficiency. It turns out that there are faster ways to sort numbers in-place, e.g. using either HeapSort or QuickSort.

## icon 4:48 Here is a subtle but interesting point to make regarding this sorting algorithm. Suppose that in the…

Codings:

● <<-  
● Description  
● In vivo term introduction

Content:

Here is a subtle but interesting point to make regarding this sorting algorithm. Suppose that in the if- statement above, we have A[i] = A[j]. Observe that in this case we copy from the left sublist. Would it have mattered if instead we had copied from the right sublist? The simple answer is no—since the elements are equal, they can appear in either order in the final sublist. However there is a subtler reason to prefer this particular choice. Many times we are sorting data that does not have a single attribute, but has many attributes (name, SSN, grade, etc.) Often the list may already have been sorted on one attribute (say, name). If we sort on a second attribute (say, grade), then it would be nice if people with same grade are still sorted by name. A sorting algorithm that has the property that equal items will appear in the final sorted list in the same relative order that they appeared in the initial input is called a stable sorting algorithm.

## icon 4:49 This is a nice property for a sorting algorithm to have. By favoring elements from the left sublist…

Codings:

● ->  
● Property

Content:

This is a nice property for a sorting algorithm to have. By favoring elements from the left sublist over the right, we will be preserving the relative order of elements. It can be shown that as a result, MergeSort is a stable sorting algorithm.

## icon 4:50 Analysis: What remains is to analyze the running time of MergeSort.

Codings:

● <->  
● Complexity

Content:

Analysis: What remains is to analyze the running time of MergeSort.

## icon 4:51 First let us consider the running time of the procedure Merge(A, p, q, r). Let n = r − p + 1 denote…

Codings:

● ->  
● Description  
● Mathematic  
● Operation

Content:

First let us consider the running time of the procedure Merge(A, p, q, r). Let n = r − p + 1 denote the total length of both the left and right subarrays. What is the running time of Merge as a function of n? The algorithm contains four loops (none nested in the other). It is easy to see that each loop can be executed at most n times. (If you are a bit more careful you can actually see that all the while-loops together can only be executed n times in total, because each execution copies one new element to the array B, and B only has space for n elements.) Thus the running time to Merge n items is Θ(n). Let us write this without the asymptotic notation, simply as n. (We’ll see later why we do this.)

## icon 4:53 the use of a recurrence, that is, a function that is defined recursively in terms of itself. To avoi…

Codings:

● <-  
● Description  
● In vivo term introduction

Content:

the use of a recurrence, that is, a function that is defined recursively in terms of itself. To avoid circularity, the recurrence for a given value of n is defined in terms of values that are strictly smaller than n. Finally, a recurrence has some basis values (e.g. for n = 1), which are defined explicitly.

Let’s see how to apply this to MergeSort. Let T (n) denote the worst case running time of MergeSort on an array of length n. For concreteness we could count whatever we like: number of lines of pseudocode, number of comparisons, number of array accesses, since these will only differ by a constant factor. Since all of the real work is done in the Merge procedure, we will count the total time spent in the Merge procedure.

First observe that if we call MergeSort with a list containing a single element, then the running time is a constant. Since we are ignoring constant factors, we can just write T (n) = 1. When we call MergeSort with a list of length n > 1, e.g. Merge(A, p, r), where r−p+1 = n, the algorithm first computes q = ⌊(p + r)/2⌋. The subarray A[p..q], which contains q − p + 1 elements. You can verify (by some tedious floor-ceiling arithmetic, or simpler by just trying an odd example and an even example) that is of size ⌈n/2⌉. Thus the remaining subarray A[q + 1..r] has ⌊n/2⌋ elements in it. How long does it take to sort the left subarray? We do not know this, but because ⌈n/2⌉ < n for n > 1, we can express this as T (⌈n/2⌉). Similarly, we can express the time that it takes to sort the right subarray as T (⌊n/2⌋). Finally, to merge both sorted lists takes n time, by the comments made above. In conclusion we have

## icon 4:55 T(n) =

Codings:

● Definition  
● Mathematic

Content:

T(n) =

## icon 4:56 Lecture 6: Divide and Conquer and MergeSort

Codings:

● Algorithm

Content:

Lecture 6: Divide and Conquer and MergeSort

# icon 7 MS02.pdf

33 Quotations:

## icon 7:1 Consider sorting the values in an array A of size N. Most sorting algorithms involve what are called…

Codings:

● ->  
● Class  
● In vivo term introduction

Content:

Consider sorting the values in an array A of size N. Most sorting algorithms involve what are called comparison

sorts; i.e., they work by comparing values.

## icon 7:2 Comparison sorts can never have a worst-case running time less than O(N log N). Simple comparison so…

Codings:

● ->  
● Complexity  
● Description

Content:

Comparison sorts can never have a worst-case running time less than O(N log N). Simple comparison sorts are usually O(N2); the more clever ones are O(N log N).

## icon 7:3 Does an algorithm always take its worst-case time? What happens on an already-sorted array? How much…

Codings:

● <->  
● Design

Content:

Does an algorithm always take its worst-case time?

What happens on an already-sorted array?

How much space (other than the space for the array itself) is required?

## icon 7:4 We will discuss four comparison-sort algorithms:

Codings:

● Comment  
● Meta

Content:

We will discuss four comparison-sort algorithms:

## icon 7:5 Selection sort and insertion sort have

Codings:

● <->  
● Algorithm

Content:

Selection sort and insertion sort have

## icon 7:7 As mentioned above, merge sort takes time O(N log N), which is quite a bit better than the two O(N2)…

Codings:

● Description

Content:

As mentioned above, merge sort takes time O(N log N), which is quite a bit better than the two O(N2) sorts described above (for example, when N=1,000,000, N2=1,000,000,000,000, and N log2 N = 20,000,000; i.e., N2 is 50,000 times larger than N log N!).

## icon 7:9 The key insight behind merge sort is that it is possible to merge two sorted arrays, each containing…

Codings:

● <-  
● Description

Content:

The key insight behind merge sort is that it is possible to merge two sorted arrays, each containing N/2 items to form one sorted array containing N items in time O(N). To do this merge, you just step through the two arrays, always choosing the smaller of the two values to put into the final array (and only advancing in the array from which you took the smaller value).

## icon 7:10 Here's a picture illustrating this merge process:

Codings:

● Cartoon  
● Example

Content:

Here's a picture illustrating this merge process:

## icon 7:11 Now the question is, how do we get the two sorted arrays of size N/2?

Codings:

● ->  
● Pedagogical  
● Problem  
● Solicitation

Content:

Now the question is, how do we get the two sorted arrays of size N/2?

## icon 7:12 The answer is to use recursion; to sort an array of length N:

Codings:

● ->  
● Solution

Content:

The answer is to use recursion; to sort an array of length N:

## icon 7:13 Divide the array into two halves. 2. Recursively, sort the left half. 3. Recursively, sort the right…

Codings:

● Description  
● Sequence

Content:

Divide the array into two halves. 2. Recursively, sort the left half.

3. Recursively, sort the right half. 4. Merge the two sorted halves.

## icon 7:14 The base case for the recursion is when the array to be sorted is of length 1 -- then it is already…

Codings:

● Cases  
● Description

Content:

The base case for the recursion is when the array to be sorted is of length 1 -- then it is already sorted, so there is nothing to do.

## icon 7:15 Note that the merge step (step 4) needs to use an auxiliary array (to avoid overwriting its values).

Codings:

● caveat

Content:

Note that the merge step (step 4) needs to use an auxiliary array (to avoid overwriting its values).

## icon 7:16 The sorted values are then copied back from the auxiliary array to the original array.

Codings:

● Description

Content:

The sorted values are then copied back from the auxiliary array to the original array.

## icon 7:17 An outline of the code for merge sort is given below. It uses an auxiliary method with extra paramet…

Codings:

● Code  
● Example

Content:

An outline of the code for merge sort is given below. It uses an auxiliary method with extra parameters that tell what part of array A each recursive call is responsible for sorting.

## icon 7:18 Fill in the missing code in the mergeSort method.

Codings:

● Meta  
● Pedagogical

Content:

Fill in the missing code in the mergeSort method.

## icon 7:19 merge sort -- that work by dividing the problem in two, solving the smaller versions, and then combi…

Codings:

● In vivo term introduction

Content:

merge sort -- that work by dividing the problem in two, solving the smaller versions, and then combining the solutions -- are called divide and conquer algorithms.

## icon 7:20 Below is a picture illustrating the divide- and-conquer aspect of merge sort using a new example arr…

Codings:

● Cartoon  
● Example

Content:

Below is a picture illustrating the divide- and-conquer aspect of merge sort using a new example array.

## icon 7:21 The picture shows the problem being divided up into smaller and smaller pieces (first an array of si…

Codings:

● Description

Content:

The picture shows the problem being divided up into smaller and smaller pieces (first an array of size 8, then two halves each of size 4, etc). Then it shows the "combine" steps: the solved problems of half size are merged to form solutions to the larger problem. (Note that the picture illustrates the conceptual ideas -- in an actual execution, the small problems would be solved one after the other, not in parallel. Also, the picture doesn't illustrate the use of auxiliary arrays during the merge steps.)

## icon 7:22 To determine the time for merge sort, it is helpful to visualize the calls made to mergeAux as shown…

Codings:

● <->  
● Cartoon  
● Complexity  
● Description

Content:

To determine the time for merge sort, it is helpful to visualize the calls made to mergeAux as shown below (each node represents one call, and is labeled with the size of the array to be sorted by that call):

## icon 7:23 The height of this tree is O(log N). The total work done at each "level" of the tree (i.e., the work…

Codings:

● ->  
● Observation  
● State

Content:

The height of this tree is O(log N). The total work done at each "level" of the tree (i.e., the work done by mergeAux excluding the recursive calls) is O(N):

## icon 7:24 Step 1 (finding the middle index) is O(1), and this step is performed once in each call; i.e., a tot…

Codings:

● Description

Content:

Step 1 (finding the middle index) is O(1), and this step is performed once in each call; i.e., a total of once at the top level, twice at the second level, etc, down to a total of N/2 times at the second-to-last level (it is not performed at all at the very last level, because there the base case applies, and mergeAux just returns). So for any one level, the total amount of work for Step 1 is at most O(N).

For each individual call, Step 4 (merging the sorted half-graphs) takes time proportional to the size of the part of the array to be sorted by that call. So for a whole level, the time is proportional to the sum of the sizes at that level. This sum is always N.

Therefore, the time for merge sort involves O(N) work done at each "level" of the tree that represents the recursive calls. Since there are O(log N) levels, the total worst-case time is O(N log N).

## icon 7:25 What happens when the array is already sorted (what is the running time for merge sort in that case)…

Codings:

● Meta  
● Pedagogical

Content:

What happens when the array is already sorted (what is the running time for merge sort in that case)?

## icon 7:26 Introduction

Codings:

● Algorithm

Content:

Introduction

## icon 7:27 case time O(N2).

Codings:

● ->  
● Complexity

Content:

case time O(N2).

## icon 7:28 Quick

Codings:

● <-

Content:

Quick

## icon 7:29 sort

Codings:

● <->  
● Algorithm

Content:

sort

## icon 7:30 O(N2) in the worst case, but its expected time is O(N log N)

Codings:

● ->  
● Complexity

Content:

O(N2) in the worst case, but its expected time is O(N log N)

## icon 7:31 Merge

Codings:

● <-

Content:

Merge

## icon 7:32 sort

Codings:

● <->  
● Algorithm

Content:

sort

## icon 7:33 O(N log N) in the worst case.

Codings:

● ->  
● Complexity

Content:

O(N log N) in the worst case.

## icon 7:34 Algorithms

Codings:

● <-

Content:

Algorithms

## icon 7:35 like

Codings:

● <->  
● Class

Content:

like

# icon 13 MS03.pdf

23 Quotations:

## icon 13:1 2.2 Mergesort

Codings:

● Algorithm

Content:

2.2 Mergesort

## icon 13:2 The algorithms that we consider in this section is based on a simple operation known as merging: com…

Codings:

● ->  
● Description  
● In vivo term introduction  
● Operation

Content:

The algorithms that we consider in this section is based on a simple operation known as merging: combining two ordered arrays to make one larger ordered array.

## icon 13:3 This operation immediately lends itself to a simple recursive sort method known as mergesort: to sor…

Codings:

● <-  
● Definition  
● Sequence

Content:

This operation immediately lends itself to a simple recursive sort method known as mergesort: to sort an array, divide it into two halves, sort the two halves (recursively), and then merge the results.

## icon 13:4 Mergesort guarantees to sort an array of N items in time proportional to N log N, no matter what the…

Codings:

● ->  
● Description  
● Mathematic  
● Motivation

Content:

Mergesort guarantees to sort an array of N items in time proportional to N log N, no matter what the input.

## icon 13:5 prime disadvantage is that it uses extra space proportional to N.

Codings:

● <->  
● Description  
● Disadvantages  
● Mathematic

Content:

prime disadvantage is that it uses extra space proportional to N.

## icon 13:6 Abstract in-place merge.

Codings:

● <->  
● Operation

Content:

Abstract in-place merge.

## icon 13:7 The method merge(a, lo, mid, hi) in Merge.java puts the results of merging the subarrays a[lo..mid]…

Codings:

● Description

Content:

The method merge(a, lo, mid, hi) in Merge.java puts the results of merging the subarrays a[lo..mid] with a[mid+1..hi] into a single ordered array, leaving the result in a[lo..hi].

## icon 13:8 While it would be desirable to implement this method without using a significant amount of extra spa…

Codings:

● Aside  
● Related

Content:

While it would be desirable to implement this method without using a significant amount of extra space, such solutions are remarkably complicated. Instead, merge() copies everything to an auxiliary array and then merges back to the original.

## icon 13:9 original

Codings:

● <-  
● Cartoon  
● Description

Content:

original

## icon 13:10 Merge.java is a recursive mergesort implementation based on this abstract in-place merge. It is one…

Codings:

● Description  
● In vivo term introduction

Content:

Merge.java is a recursive mergesort implementation based on this abstract in-place merge. It is one of the best-

known examples of the utility of the divide-and-conquer paradigm for efficient algorithm

## icon 13:11 design

Codings:

● Cartoon  
● Description

Content:

design

## icon 13:12 Top-down mergesort uses between 1/2 N lg N and N lg N compares and at most 6 N lg N array accesses t…

Codings:

● ->  
● Complexity  
● Proposal

Content:

Top-down mergesort uses between 1/2 N lg N and N lg N compares and at most 6 N lg N array accesses to sort

## icon 13:13 We can cut the running time of mergesort substantially with some carefully considered modifications…

Codings:

● <->  
● Implementation

Content:

We can cut the running time of mergesort substantially with some carefully considered modifications to the implementation.

## icon 13:14 Use insertion sort for small subarrays.

Codings:

● ->  
● Design

Content:

Use insertion sort for small subarrays.

## icon 13:15 We can improve most recursive algorithms by handling small cases differently. Switching to insertion…

Codings:

● Description

Content:

We can improve most recursive algorithms by handling small cases differently. Switching to insertion sort for small subarrays will improve the running time of a typical mergesort implementation by 10 to 15 percent.

Test whether array is already in order. We can reduce the running time to be linear for arrays that are already in order by adding a test to skip call to merge() if a[mid] is less than or equal to a[mid+1]. With this change, we still do all the recursive calls, but the running time for any sorted subarray is linear. Eliminate the copy to the auxiliary array. It is possible to eliminate the time (but not the space) taken to copy to the auxiliary array used for merging. To do so, we use two invocations of the sort method, one that takes its input from the given array and puts the sorted output in the auxiliary array; the other takes its input from the auxiliary array and puts the sorted output in the given array. With this approach, in a bit of mindbending recursive trickery, we can arrange the recursive calls such that the computation switches the roles of the input array and the auxiliary array at each level.

## icon 13:16 MergeBars.java provides a visualization of mergesort with cutoff for small subarrays.

Codings:

● <-  
● Cartoon

Content:

MergeBars.java provides a visualization of mergesort with cutoff for small subarrays.

## icon 13:17 Bottom-up mergesort.

Codings:

● Algorithm  
● Related

Content:

Bottom-up mergesort.

## icon 13:18 Even though we are thinking in terms of merging together two large subarrays, the fact is that most…

Codings:

● Description

Content:

Even though we are thinking in terms of merging together two large subarrays, the fact is that most merges are merging together tiny subarrays. Another way to implement mergesort is to organize the merges so that we do all the merges of tiny arrays on one pass, then do a second pass to merge those arrays in pairs, and so forth, continuing until we do a merge that encompasses the whole array. This method requires even less code than the standard recursive implementation. We start by doing a pass of 1-by-1 merges (considering individual items as subarrays of size 1), then a pass of 2-by-2 merges (merge subarrays of size 2 to make subarrays of size 4), then 4-by-4 merges, and so forth. MergeBU.java is an implementation of

## icon 13:19 bottom-up mergesort.

Codings:

● Cartoon  
● Description

Content:

bottom-up mergesort.

## icon 13:20 Bottom-up mergesort uses between 1/2 N lg N and N lg N compares and at most 6 N lg N array accesses…

Codings:

● ->  
● Complexity

Content:

Bottom-up mergesort uses between 1/2 N lg N and N lg N compares and at most 6 N lg N array accesses to sort

## icon 13:21 No compare-based sorting algorithm can guarantee to sort N items with fewer than lg(N!) ~ N lg N com…

Codings:

● Proposal

Content:

No compare-based sorting algorithm can guarantee to sort N items with fewer than lg(N!) ~ N lg N compares.

## icon 13:22 Mergesort is an asymptotically optimal compare-based sorting algorithm. That is, both the number of…

Codings:

● Description  
● Proposal

Content:

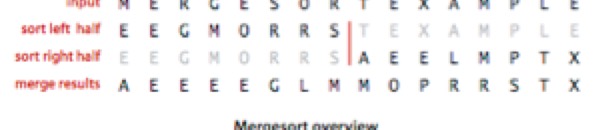
Mergesort is an asymptotically optimal compare-based sorting algorithm. That is, both the number of compares used by mergesort in the worst case and the minimum number of compares that any compare-based sorting algorithm can guarantee are ~N lg N.

## icon 13:23 Quotation 13:23

Codings:

● Cartoon

Content:



# icon 10 MS04.pdf

44 Quotations:

## icon 10:1 Contents:

Codings:

● Algorithm  
● Outline

Content:

Contents:

## icon 10:2 Our next algorithm actually achieves the optimal big-O behavior for a sorting algorithm. The merge s…

Codings:

● ->  
● Description  
● Motivation

Content:

Our next algorithm actually achieves the optimal big-O behavior for a sorting algorithm. The merge sort has time for both its worst and average case.

## icon 10:3 This doesn’t necessarily make it the ideal choice, however, in all sorting applications. The constan…

Codings:

● caveat  
● Description

Content:

This doesn’t necessarily make it the ideal choice, however, in all sorting applications. The constant multiplier on the timing is somewhat high, and merge sort may require an unusually high amount of memory.

## icon 10:4 Variants of the basic merge sort algorithm are, however, often used with linked lists (which can’t b…

Codings:

● Aside  
● Description

Content:

Variants of the basic merge sort algorithm are, however, often used with linked lists (which can’t be sorted by most other algorithms and are used to sort data residing on disk or magnetic tape.

## icon 10:5 Before tackling the merge sort itself, we start with a simpler function that is used by merge sort.

Codings:

● <->  
● Operation

Content:

Before tackling the merge sort itself, we start with a simpler function that is used by merge sort.

## icon 10:6 Suppose that our sequence of data can be divided into two parts, such that a[first..mid-1] is alread…

Codings:

● Description

Content:

Suppose that our sequence of data can be divided into two parts, such that a[first..mid-1] is already sorted and a[mid..last-1] is already sorted. Then we could merge the two parts into a combined sorted sequence using the code shown here.

## icon 10:7 template <typename T>

Codings:

● Code  
● Definition

Content:

template <typename T>

## icon 10:8 The heart of the merge algorithm is the first loop (\co1). The variables first, mid, and last mark o…

Codings:

● Description

Content:

The heart of the merge algorithm is the first loop (\co1).

The variables first, mid, and last mark off two subsequences that we want to merge. We can think of a[first ... mid-1] and a[mid ... last-1] as two separate, sorted sequences. We want to combine them into a single sorted sequence, tempVector.

The way to do this is quite simple. Just compare the first element in each of the two input (sub)sequences and copy the smaller one.

For example, if we were merging subsequences [ 2 \; 4 \; 5 \; 6 ] and [ 1 \; 3] we would compare the first element in each one (2 and 1) and decide to copy 1.

Then we continue with the remainder, merging [ 2 \; 4 \; 5 \; 6] and [ 3 ]

On the next step we would copy 2, and be left with the merge of [ 4 \; 5 \; 6 ] and [ 3 ]

We would then copy 3.

At this point, our temporary vector contains [ 1 \; 2 \; 3] We would now exit from this main loop, because one of the arrays has been completely emptied out.

The rest of the algorithm is “cleanup”. We exit the main loop when we have emptied one of the two subsequences, so there is a possibility that the other subsequence still has data. The next two loops (\co2) copy that data from the remainder of the two subsequences. (Because one of those subsequences has been emptied, one of these loops will execute zero times.)

## icon 10:9 1.2 Merge Analysis

Codings:

● ->  
● Complexity

Content:

1.2 Merge Analysis

## icon 10:10 There are several vector push\_back calls, which have a worst-case behavior proportional to the size…

Codings:

● ->  
● Constituent  
● Description

Content:

There are several vector push\_back calls, which have a worst-case behavior proportional to the size of the vector. However, a little time looking at them shows that they are all on tempVector, which is initially empty.

## icon 10:11 So this falls into the special case pattern of filling an initially empty vector with repeated push\_…

Codings:

● Description

Content:

So this falls into the special case pattern of filling an initially empty vector with repeated push\_backs, in which case those pushes will amortize to .

## icon 10:12 This means that all the loop bodies amortize to .

Codings:

● <->  
● Constituent  
● Description

Content:

This means that all the loop bodies amortize to .

## icon 10:13 Looking at the code for the first 3 loops, note that each one adds one element into tempVector. no e…

Codings:

● <-  
● Observation

Content:

Looking at the code for the first 3 loops, note that

each one adds one element into tempVector.

no element is copied multiple times. If we copy the element at indexA, we also increment indexA, so we will not copy that element again. Similarly, if we copy the element at indexB, we also increment indexB, so we will not copy that element agai

## icon 10:14 Since there are a total of last-first elements, each loop can repeat no more than last-first times.…

Codings:

● Observation

Content:

Since there are a total of last-first elements, each loop can repeat no more than last-first times. In fact, the sum of the number of iterations of all three loops is last-first.

So all three loops are .

## icon 10:15 emplate <typename T>

Codings:

● Code  
● Definition  
● Example

Content:

emplate <typename T>

## icon 10:16 The last loop clearly repeats once for each element in tempVector. But we have just determined that…

Codings:

● Observation

Content:

The last loop clearly repeats once for each element in tempVector. But we have just determined that the total number of elements in tempVector will be last-first.

## icon 10:17 merge3.cpp

Codings:

● Code  
● Definition  
● Example

Content:

merge3.cpp

## icon 10:18 That leaves only a handful of O(1) statements that will all be dominated by the complexity of the lo…

Codings:

● ->  
● Complexity  
● Conclusion

Content:

That leaves only a handful of O(1) statements that will all be dominated by the complexity of the loops, so merge is .

## icon 10:19 sorted sequences!

Codings:

● ->  
● Description  
● Operation

Content:

sorted sequences!

## icon 10:20 But how do we get the two sorted sequences in the first place?

Codings:

● Description  
● Problem

Content:

But how do we get the two sorted sequences in the first place?

## icon 10:21 By merge’ing two even

Codings:

● ->  
● Description  
● Solution

Content:

By merge’ing two even

## icon 10:22 template <typename T>

Codings:

● Algorithm  
● Code  
● Definition

Content:

template <typename T>

## icon 10:23 The algorithm shown here is the actual sorting algorithm. It is almost amazingly simple, consisting…

Codings:

● Description

Content:

The algorithm shown here is the actual sorting algorithm. It is almost amazingly simple, consisting simply of two recursive calls to itself, each attempting to sort half the vector, followed by a call to merge to combine the two sorted halves into a single sorted sequence.

## icon 10:24 For many people, the very simplicity of this algorithm makes it hard to believe that it can work. I…

Codings:

● Aside  
● Description  
● Meta

Content:

For many people, the very simplicity of this algorithm makes it hard to believe that it can work. I therefore recommend strongly that you run this algorithm until you are comfortable with your understanding of it.

## icon 10:25 2.2 MergeSort Analysis

Codings:

● ->  
● Complexity

Content:

2.2 MergeSort Analysis

## icon 10:26 Each call to mergeSort is either done in O(1) time (if first+1 \ensuremath{\geq last}) or splits the…

Codings:

● Description

Content:

Each call to mergeSort is either done in O(1) time (if first+1 \ensuremath{\geq last}) or splits the array into two equal (\ensuremath{\pm}1) pieces. We can do this split up to log N times.

## icon 10:27 We can do this split up to log N times.

Codings:

● Cartoon  
● Example

Content:

We can do this split up to log N times.

## icon 10:28 We can envision the recursive mergeSort calls (in blue) and the subsequent calls to merge (in yellow…

Codings:

● Cartoon  
● Legend

Content:

We can envision the recursive mergeSort calls (in blue) and the subsequent calls to merge (in yellow) as a tree-like structure.

## icon 10:29 Let denote the total number of elements being sorted (the value of last-first on the very first call…

Codings:

● Definition  
● Mathematic

Content:

Let denote the total number of elements being sorted (the value of last-first on the very first call to mergeSort). \ePicOnRight

## icon 10:30 Each level in the tree involves no more than objects, split in various ways and needing to be merged…

Codings:

● Cartoon  
● Observation

Content:

Each level in the tree involves no more than objects, split in various ways and needing to be merged.

## icon 10:31 merge is , where is the number of elements to be merged. The sum of all the values at any level of t…

Codings:

● ->  
● Description  
● Operation

Content:

merge is , where is the number of elements to be merged. The sum of all the values at any level of the yellow tree is .

## icon 10:32 Consequently the combined set of merges at each level of the tree is .

Codings:

● <-  
● Conclusion

Content:

Consequently the combined set of merges at each level of the tree is .

## icon 10:33 The blue tree represents all the non-merge work in mergeSort.

Codings:

● Cartoon  
● Legend

Content:

The blue tree represents all the non-merge work in mergeSort.

## icon 10:34 But there’s only those blue nodes. Since the most blue nodes we could have at one level is , each bl…

Codings:

● Cartoon  
● Observation

Content:

But there’s only

those blue nodes. Since the most blue nodes we could have at one level is , each blue level is, at most,

work in each of

total work.

## icon 10:35 Because we have levels, each level taking work, the overall merge sort code is (worst & average case…

Codings:

● Conclusion

Content:

Because we have levels, each level taking work, the overall merge sort code is (worst & average case) .

## icon 10:36 So merge sort is as fast as any pairwise-comparison sort can be.

Codings:

● Advantages

Content:

So merge sort is as fast as any pairwise-comparison sort can be.

## icon 10:37 Still, merge sort is not considered to be the “ideal” sorting algorithm. Its primary drawbacks are

Codings:

● <->  
● Disadvantages

Content:

Still, merge sort is not considered to be the “ideal” sorting algorithm. Its primary drawbacks are

## icon 10:38 It requires extra storage (for the tempVector) It does the full set of comparisons and copies even w…

Codings:

● Description

Content:

It requires extra storage (for the tempVector)

It does the full set of comparisons and copies even when applied to arrays that are already sorted.

## icon 10:39 On the other hand, merge sort has an

Codings:

● ->  
● Operation

Content:

On the other hand, merge sort has an

## icon 10:40 The merge routine itself moves sequentially through its working arrays, not jumping from place to pl…

Codings:

● Description

Content:

The merge routine itself moves sequentially through its working arrays, not jumping from place to place. This behavior would be absolutely wonderful if we were storing our arrays in some strange kind of memory where moving forward one place is cheap, but jumping to an arbitrary position is expensive.

## icon 10:41 In fact, that “strange kind of memory” does exist: disk drives and magnetic tape both meet that desc…

Codings:

● ->  
● Application  
● Example  
● In vivo term introduction

Content:

In fact, that “strange kind of memory” does exist: disk drives and magnetic tape both meet that description. Hence variations of merge sort have long been the algorithm of choice in external sorting, sorting sets of material stored in disk/tape files that are too large to load into memory.

## icon 10:42 merge2

Codings:

● <-

Content:

merge2

## icon 10:43 cpp

Codings:

● <-

Content:

cpp

## icon 10:44 advantage that may, at first glance, not have seemed very important.

Codings:

● ->  
● Advantages

Content:

advantage that may, at first glance, not have seemed very important.

# icon 15 MS05.pdf

23 Quotations:

## icon 15:1 In lecture 6, we saw three algorithms for sorting a list of n items.

Codings:

● ->  
● Algorithm  
● Description  
● Review

Content:

In lecture 6, we saw three algorithms for sorting a list of n items.

## icon 15:2 We saw that, in the worst case, all of these algorithm required O(n2) operations.

Codings:

● ->  
● Complexity  
● Description

Content:

We saw that, in the worst case, all of these algorithm required O(n2) operations.

## icon 15:3 Such algorithms will be unacceptably slow if n is large.

Codings:

● Conclusion

Content:

Such algorithms will be unacceptably slow if n is large.

## icon 15:4 To make this claim more concrete, consider that if n = 220 ≈ 106 i.e one million, then n2 ≈ 1012.

Codings:

● Example

Content:

To make this claim more concrete, consider that if n = 220 ≈ 106 i.e one million, then n2 ≈ 1012.

## icon 15:5 How long would it take a program to run that many instructions?

Codings:

● Solicitation

Content:

How long would it take a program to run that many instructions?

## icon 15:6 Today’s processors run at about 109 basic operations per second (i.e. GHz). So a problem that takes…

Codings:

● Description

Content:

Today’s processors run at about 109 basic operations per second (i.e. GHz). So a problem that takes in the order of 1012 operations would require thousands of seconds of processing time. Having a multicore machine with say 4 processors only can speed things up by a factor of 4 – max! – which doesn’t change the argument here.

## icon 15:8 Here is the idea. If the list has just one number (n = 1), then do nothing. Otherwise, partition the…

Codings:

● <<-  
● Description

Content:

Here is the idea. If the list has just one number (n = 1), then do nothing. Otherwise, partition the list of n elements into two lists of size about n/2 elements each, sort the two individual lists (recursively, using mergesort), and then merge the two sorted lists.

## icon 15:9 For example, suppose we have a list

Codings:

● Cartoon  
● Example

Content:

For example, suppose we have a list

## icon 15:10 Here is pseudocode for the algorithm. Note that it uses a helper method merge which in fact does mos…

Codings:

● PseudoCode

Content:

Here is pseudocode for the algorithm. Note that it uses a helper method merge which in fact

does most of the work.

## icon 15:11 Here is the merge algorithm.

Codings:

● ->  
● Operation

Content:

Here is the merge algorithm.

## icon 15:12 Note that it has two phases. The first phase initializes a new list (empty), steps through the two l…

Codings:

● Description

Content:

Note that it has two phases. The first phase initializes a new list (empty), steps through the two lists, (list1) and (list2), compares the front element of each list and removes the smaller of the two, and this removed element to to the back of the merged list. See the detailed example in the slides for an illustration.

The second phase of the algorithm starts after one of list1 or list2 becomes empty. In this case, the remaining elements from the non-empty list are moved to list. This second phase uses two while loops in the above pseudocode, and note that only one of these two loops will be used since we only reach phase two when one of list1 or list2 is already empty.

## icon 15:13 merge( list1, list2){

Codings:

● Definition  
● PseudoCode

Content:

merge( list1, list2){

## icon 15:14 I have written the mergesort and merge algorithms using abstract list operations only, rather than s…

Codings:

● <->  
● Design  
● Observation

Content:

I have written the mergesort and merge algorithms using abstract list operations only, rather than specifying how exactly it is implemented (array list versus linked list). Staying at an abstract level has the advantage of getting us quickly to the main ideas of the algorithm:

## icon 15:15 what is being computed and in which sequence?

Codings:

● Solicitation

Content:

what is being computed and in which sequence?

## icon 15:16 However, be aware that there are disadvantages of hiding the implementation details, i.e. the data s…

Codings:

● Comment  
● Meta  
● Pedagogical

Content:

However, be aware that there are disadvantages of hiding the implementation details, i.e. the data structures. As we have seen, sometimes the choice of data structure can be important for performance.

## icon 15:17 For example, compare an array versus a linked list implementation. The call getElements() within mer…

Codings:

● Description

Content:

For example, compare an array versus a linked list implementation. The call getElements() within mergesort would be different for these two data structures. For the array, getElements() might just compute the start and end indices for the two lists. These indices could be passed as parameters to the mergesort calls. For a linked list, it would be necessary to iterate through the list to find the location of the mid element, and one could then break up the list into the list1 and list2 with the mid list element being at the tail of list1 and the mid + 1 element being at the head of list2.

For the merge, if one were using an array, then one could use a second array (list) as a buffer for doing the merges. One could copy the elements from list1 and list2 to this second array. At the next level of the recursion, one could copy back to the first array, and go back and forth. So the space requirements would be double the size of the original list.

## icon 15:18 mergesort is O(n log n)

Codings:

● <->  
● Complexity

Content:

mergesort is O(n log n)

## icon 15:19 There are log n levels of the recursion, namely the number of levels is the number of times that you…

Codings:

● Description

Content:

There are log n levels of the recursion, namely the number of levels is the number of times that you

can divide the list size n by 2 until you reach 1 element per list. The number of instructions that

## icon 15:20 must be executed at each level of the recursion is proportional to the number n of items in the list…

Codings:

● Description  
● In vivo term introduction

Content:

must be executed at each level of the recursion is proportional to the number n of items in the list. Thus, the total number of instructions is proportional to n ∗ log2 n, or as usually written n log2 n. I will discuss this again a few lectures from now, when we study recurrences.

## icon 15:21 To appreciate the difference between the number of operations for the earlier O(n2) sorting algorith…

Codings:

● Example  
● Table

Content:

To appreciate the difference between the number of operations for the earlier O(n2) sorting algorithms versus O(n log n) for mergesort, consider the following table.

## icon 15:22 Thus, the time it takes to run mergesort is significantly less than the time it takes to run bub- bl…

Codings:

● Description

Content:

Thus, the time it takes to run mergesort is significantly less than the time it takes to run bub- ble/selection/insertion sort, when n becomes large. Very roughly speaking, on a computer that runs 109 operations per second running mergesort on a list of size n = 109 would take in the order of minutes, whereas running insertion sort would take centuries. (After class, one student asked me how I came up with such time estimates. That’s easy: just consider there are 60 seconds/minute, 60 minutes/hour, 24 hours/day, etc.)

## icon 15:23 In the lectures slides, I went over an examples of how the various calls to mergesort and merge work…

Codings:

● <-  
● Comment  
● Meta  
● Pedagogical  
● Solicitation

Content:

In the lectures slides, I went over an examples of how the various calls to mergesort and merge work. The tricky part is to see the order of the various recursive calls and exits. It is easy to understand this with pictures, so please see the slides. (We will many more examples of recursive algorithm later in the course when we look at trees and graphs, so if you don’t yet get this, then please be patient with yourself – you will!)

## icon 15:24 Mergesort

Codings:

● Algorithm

Content:

Mergesort