A black text on a white background

Description automatically generated

**Bit Manipulation**

Assume int a = 11 is a signed integer

Example: -11 ≫ 1

|  |  |  |
| --- | --- | --- |
|  | 00001011 | 11 in binary |
|  | 11110100 | Bit inversion |
| + | 1 | 2’s complement |
|  | 11110101 | -11 in binary |
|  | ≫ 1 | Right shift |
|  | 11111010 | Right bit back🡪front |
|  | 00000101 | Bit inversion |
| + | 1 | 2’s complement |
|  | 00000110 | -6 is the answer |

int a = 11;

cout ≪ (((a ≪ 3) | 14) ≫ (a & 23)) ≪ endl;

1. 11 ≪ 3 = 11 \* 23 = 88

2. Convert 88 and 14 to binary to do bitwise

|  |  |
| --- | --- |
| 01011000 | 88 in binary |
| **OR** 00001110 | 14 in binary |
| 01011110 | If 88 or 14 have 1 then 1, otherwise 0 |

3. 11 & 23

|  |  |
| --- | --- |
| 00001011 | 11 in binary |
| **OR** 00010111 | 23 in binary |
| 00000011 | If 11 and 23 have 1 then 1, otherwise 0 |

4. 94 ≫ 3

|  |  |
| --- | --- |
| 1011110 | 94 in binary |
| ≫ 3 | Right shift by 3 |
| 0001011 | 11 is the answer |

**Decimal 🡪 Binary (key #s: 128, 64, 32, 16, 8, 4, 2, 1)**

Ex: Convert 14 to binary. (Answer: 1110)

Closest value to 14 is **8**, so 14 – 8 = 6

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 0 | 0 | 0 | 1 | - | - | - |

Closest value to 6 is **4**, so 6 – 4 = 2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | - | - |

Closest value to 2 is 2, so 2 – 2 = 0 and **done**.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |

**Problem Sizes**

We have some input data of size n which we want to process using an algorithm that has a running time of f(n), expressed in microseconds. What’s the biggest n that we can process in time t? (if milliseconds, 1000 milliseconds/sec)

Ex: t = 1 hr 🡪 3600 sec/hr \* 1000000 microsec/sec = 3.6 \* 109 microseconds

If f(n) = lg(n) = 3.6 \* 109 then n = 23.6 \* 10^9

If f(n) = n = 3.6 \* 109

If f(n) = n lg n = 133378058 (using binary search)

**Binary Search C++ Code**

#include <iostream>

#include <cmath>

const double epsilon = 1e-9;

// Iterative version

int nlogn(double c) {

double lower = 0.0;

double upper = c;

while (true) {

double middle = lower + (upper - lower) / 2;

double val = middle \* log2(middle);

if (std::abs(c - val) <= epsilon) {

// Rounding down to integer

return static\_cast<int>(middle);

}

if (val > c) {

upper = middle;

} else {

lower = middle;

}

}

}

// Recursive version

int nlogn\_rec\_helper(double c, double lower, double upper) {

double middle = lower + (upper - lower) / 2;

double val = middle \* log2(middle);

if (std::abs(c - val) <= epsilon) {

return static\_cast<int>(middle);

}

if (val > c) {

return nlogn\_rec\_helper(c, lower, middle);

} else {

return nlogn\_rec\_helper(c, middle, upper);

}

}

int nlogn\_rec(double c) {

return nlogn\_rec\_helper(c, 0.0, c);

}

**Proving Efficiency Classes**

f(n) = n + n ln(n) and g(n) = n√n, f ∈ O(g) because g(n) grows faster.

For f(n) = n and g(n) = 2lg(n), g ∈ O(f) because g(n) simplifies to constant time and f grows faster.

Prove f(n) = 7n2 + 4n + 3 ∈ Ω(n3).

Assume f(n) = 7n2 + 4n + 3 ∈ Ω(n3). We need to find constants c and n such that f(n) > cn3 ∀n > k. Let’s say c = 1. f(n) = 7n2 + 4n + 3 < 1\*n3. Since this is the smallest integer that could make this true, there do not exist any constants to make this true **∴** f(n) ∉ Ω(n3)

**Recurrence Relations**

General form of recurrence relations: T(n) = aT(n-b) + f(n)

a: # of times the recursive function calls itself

n – b: means this is a decrease-and-conquer algo, where the input size decreases in fixed steps of size b

b: the amount by which the input data is decreased in a recursive call

f(n): amount of work performed in the code of the function excluding recursive calls

**Solving Recurrence Relations**

1. replace n with n-1 2. replace n with n-2 3. make general form

4. initial condition 5. use sum formula on the left side to make x(n)

**≪ left shift**

**≫ right shift**

**Unsigned ints:**

- Left shift by n = multiplying the original int by 2n

- Right shift by n = integer division by 2n

**Signed ints:**

- Left shift by n = integer multiplication by 2n

- Right shift by n: no shortcut, see example

**Running Times**

**Selection Sort** (iterative): Θ(n2), (recursive): Θ(n2)

**Binary Search** (iterative): O(lg n), (recursive): O(lg n), best case: O(1) when central index matches the value we are searching for.

**Bit Manipulation** (unique letters) O(n2), using hash table to keep track of which characters we have already seen O(n), using Booleans to keep track of characters we have seen Θ(1), overall O(n)

**BubbleSort BubbleSortOpt SelectionSort InsertionSort**

best: Θ(n²) Θ(n) Θ(n²) Θ(n)

worst: Θ(n²) Θ(n²) Θ(n²) Θ(n²)

average: Θ(n²) Θ(n²) Θ(n²) Θ(n²)

overall: Θ(n²) O(n²) Θ(n²) O(n²)

Insertion sort is best of all because shifting is faster than swapping and a pass can be stopped early if the element we are trying to insert is close to or at the position where it should be. (like in the case of i=3)

Bubble sort and bubble sort optimized are worse than selection sort.

**BFS adjacency matrix**: θ(V2), adjacency list θ(V+E)

**DFS adjacency matrix**: θ(V2), adjacency list θ(V+E)

**BFS**

- Every vertex goes in and out of the queue exactly 1 time because array keeps track of whether vertices have been visited. Updating counter & array also happens once. This will all take θ(V) operations

- For every vertex, we need to find neighbors in the inner for loop

- Matrix: θ(V) operations for each vertex so θ(V2) for all vertices

- List: θ(d) operations for each vertex, θ(V+E) for all vertices

- Total # of operations

- Matrix: θ(V2)

- List: θ(V+E)

**DFS**

- There is exactly one function call and one function return for every vertex. Counter and array are also updated exactly once for each vertex. This makes a total of θ(V) operations

- For every vertex, we need to find neighbors in the inner for loop

- Matrix: θ(V) operations for each vertex so θ(V2) for all vertices

- List: θ(d) operations for each vertex, θ(V+E) for all vertices

- Total # of operations

- Matrix: θ(V2)

- List: θ(V+E)

**Topological sort**:

- Using an adjacency matrix:  
 - Initializing the array: Θ(V²)  
 - Initializing the set S: Θ(V)  
 - Managing S and L: Θ(V)  
 - Managing the array: Θ(E)  
 - Finding adjacent vertices: Θ(V²)  
 - Checking for success: O(V)  
- Using an adjacency list:  
 - Initializing the array: Θ(V + E)  
 - Initializing the set S: Θ(V)  
 - Managing S and L: Θ(V)  
 - Managing the array: Θ(E)  
 - Finding adjacent vertices: Θ(V + E)  
 - Checking for success: O(V)  
So we get the following running times for Kahn's Algorithm:  
 - Using an adjacency matrix: Θ(V²).  
 - Using an adjacency list: Θ(V + E).

**Quicksort**: Insertion sort > quicksort on smaller input sizes

best: θ(n lg n)

worst: θ(n2)

average: θ(n lg n)

**Mergesort** (best, worst, overall, average): θ(n lg n)

Counting inversions: θ(n lg n)

Max num of inversions in an array: θ(n2)

Mergesort does not use θ(1) extra memory in the best case

**Quickselect**:

best: θ(n)

worst: θ(n2)

average: θ(n)

**Radix Sort**:

θ(n) runtime

**2-3 Trees**

insert, delete, search: O(tree height) = O(lg(n)) In what region of memory

has the following Student object been created? Student (“John”, “Doe”);

**STACK**

**Floyd’s Algorithm**: computing a path that exists O(V)

A close up of text

Description automatically generated

**Huffman Encoding**

given:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.25 | 0.1 | 0.15 | 0.15 | 0.3 |
| A | B | C | D | E |

1. Sort current data

2. combine 2 smallest values

3. traverse the tree. L = 0, right = 1

**Prim’s**

- Using an Adjacency matrix (with weights) + an unordered array for the

remaining vertices: we have to repeatedly search in the array for the

next vertex to remove, so Θ(V²) in total.

- Using an Adjacency list (with weights) + a priority queue implemented as

a min-heap (where insert / delete / etc. operations take O(log n)

time): we have to look at all the neighbors of each vertex being

removed from the list of remaining vertices, so in total we have to

look at all the edges in the graph, and for each of these we might have

to update the priority queue, so O(E log V) in total.

**Krushkal’s**

Using MergeSort, the running time is then Θ(E log E).

insertion sort: θ(E2)

selection sort: θ(E2 + E lg V)

bubble sort: θ(E2 + E lg V)

quicksort: θ(E lg E + E lg V)

radixsort: θ(k\*E + E lg V)

counting sort: θ(E+K), K=range of integers

heap sort: θ(E lg E)

**Max flow**

- We find the next augmenting path in the residual network using

Breadth-First Search. I.e. we select the shortest path (in terms of

number of edges) from the source s to the sink t as the augmenting path.

The idea is that, by using the shortest path, we try to maximize the

efficiency of the algorithm.

- Running time is O(VE²).

- Edmonds-Karp uses BFS to select the augmenting path, resulting in an algorithm that runs in "only" O(VE²).

- Edmonds-Karp is not the fastest algorithm known today. There is

a more complex algorithm that can solve the Maximum Flow problem in O(VE).

There is even a very recent (and very complex) algorithm that can solve the

problem in almost linear time in the number of edges O(E^(1+ε)) (for ε as

small as you want) but unfortunately it only works for gigantic networks.

**Argc and argv**

argv[0] 🡪 program name

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| m | a | x | t | w | o |

argv[1] 🡪 23 is the first CLA

|  |  |
| --- | --- |
| 2 | 3 |

argv[2] 🡪 57 is the second CLA

|  |  |
| --- | --- |
| 5 | 7 |

argc = 3: 1 for program name, 2 for CLAs

**Pointers – pass by value**

DEFINE: void pass\_by\_value(int k)

CALL: pass\_by\_value(x)

|  |  |
| --- | --- |
| 0992 |  |
| 1000 | 5 |
| 1008 |  |
| 1992 |  |
| 2000 | 5 |
| 2008 |  |

**Example**:

void add25(int d) {

d = d + 25;

}

int main() {

int a = 10;

add25(a);

cout << a;

return 0;

}

**Pointers – pass by pointer**

DEFINE: void pass\_by\_pointer(int \*k)

CALL: pass\_by\_value(&x)

|  |  |
| --- | --- |
| 0992 |  |
| 1000 | 2000 |
| 1008 |  |
| 1992 |  |
| 2000 | 5 |
| 2008 |  |

**Example:**

void add25(int \*d) {

\*d = \*d + 25;

}

int main() {

int a = 10;

add25(&a);

cout << a;

return 0;

}

**Pointers – pass by reference**

DEFINE: void pass\_by\_pointer(int &k)

CALL: pass\_by\_value(x)

**Example:**

void add25(int &d) {

d = d + 25;

}

int main() {

int a = 10;

add25(a);

cout << a;

return 0;

}

**Newton’s Method: calculate w/0.00001 accuracy**

|  |  |  |
| --- | --- | --- |
| Formula: 1/2 \* (lastGuess + (n / lastGuess)) | | |
| **lastGuess** | **nextGuess** | **difference** |
| 1 | ½ \* (1+(2/1)) = 1.5 | 0.5 |
| 1.5 | ½ \* (1.5+(2/1.5)) = 1.41666 | 0.08333 |
| 1.141667 | ½ \* (1.141667+(2/1.141667) = 1.4142156 | 0.0024 |
| 1.4142156 | ½ \* (1.4142+(2/1.4142) = 1.414213562 | 0.0000021 |

Since difference < accuracy requirement, we are done

**Sieve algorithm**

Input: integer n > 1

Let is\_prime be an array of bool values, indexed by integers 2 to n, initialized to true

for i = 2, 3, 4, ..., while i ≤ √n:

if is\_primse[i] is true:

for j = i2, i2 + 1, i2 + 2i, ..., while j ≤ n:

is\_primse[j] = false

Now all i | **is\_prime[i] = true** are prime

**Orders of Growth**

A graph with lines and numbers

Description automatically generated

A graph of a function

Description automatically generatedA graph of a function

Description automatically generated

A graph of a function

Description automatically generated

**Find c and n0**

Sum formula for n integers: **n(n+1)/2**

Prove that f(n) = 2n2 + 4n + 12 ∈ O(n2).

1. Ignore all lower-order terms f(n) ≤ 2n2 + 4n

2. Compare with standard form O(n2) = cn2

3. Find constants f(n) ≤ cn2 ∀ n ≥ n0 2n2 + 4n ≤ **3**n2

4. Solve for n0 4n ≤ n2 🡪 4 ≤ n

**∴ c = 3, n0 = 4**

void display(const vector<MyPoint> &points) {

for(auto it = points.**cbegin**(); it != points.**cend**(); it++) {

it 🡪 print\_coords();

}

}

**Accessing array elements (dynamic memory)**

using array notation (i.e. values[0]) or pointer arithmetic (which C++ compiler automatically does)

**Deallocating memory**

delete[] values;

**Bitwise Operations**

& AND | OR

^ Exclusive OR ~ NOT

13 & 7

00001101 00001101

**&** 00000111 **|** 00000111

00000101 00001111

00001101

**^** 00000111 **~** 00001101

00001010 11110010

00001101

+ 1

00001110 = -14

**In general, ~x = -x - 1**

k

HARD COPY – ALTERING X WILL NOT CHANGE K

x

Copy but can’t modify each other

main 🡪

add25 🡪

|  |  |
| --- | --- |
| a | 10 |
| d | 10 |

|  |  |
| --- | --- |
| a | 10 |
| d | 10🡪35 |

|  |  |
| --- | --- |
| a | 10 |
| d | 35 |

main 🡪

add25 🡪

add25 function ends, clean up stack

main 🡪

add25 🡪

**Dynamic memory**

The values pointer is dynamically allocated from heap onto stack using **new** operator

int \*values = new int[x]

|  |  |
| --- | --- |
| 0992 |  |
| 1000 | 34992 |
| ... | |
| 34992 | values+1 |
| 35000 | values+2 |
| 35008 | values+3 |

🡨 values

points at

Output: 10

To convert **binary 🡪 decimal**, add up the values that have a 1 in the table. For 14, **2 + 4 + 8 = 14**

k

points at

x

Implementing topological sort using DFS is possible and it only takes θ(V) operations so running time is same as DFS itself

Correctness check (doesn’t exist) but if we wanted to add it, storing the extra array in final position will take θ(V) operations, going through each edge and checking is θ(V2) for matrix & θ(V+E) for list

Output: 35

|  |  |
| --- | --- |
| a | 10 |
| ? | 0x27f3 |

|  |  |
| --- | --- |
| a | 10 |
| \*d | 0x27f3 |

|  |  |
| --- | --- |
| a | 10🡪35 |
| \*d | 0x27f3 |

|  |  |
| --- | --- |
| a | 35 |
| \*d | 0x27f3 |

main 🡪

add25 🡪

main 🡪

add25 🡪

main 🡪

add25 🡪

🡨 address of a

the value that the pointer points to +25

**Binary Search Example**

Search 23

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 5 | 8 | 12 | 16 | 23 | 38 | 56 | 72 |

Split in 1/2 and compare 16 to 23. Since 16 < 23

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 38 | 56 | 72 | 91 |

Compare. 56 > 23 so take L side of array

|  |  |
| --- | --- |
| 23 | 38 |

Found 23 so done.

**Lomuto**:

The for loop traverses the whole array, so θ(n) for partitioning whole array

**Hoare**:

average: O(n lg n)

worst: O(n2)

best: O(n lg n)

add25 function ends, clean up stack

void func3(int n) {

if(n < 1) { return; }

for(int i = 1; i ≤ n; i++) {

for(int j = i+1; j ≤ n; j++) {

for (int k = 1; k ≤ n; k \*= 2) {

cout ≪ “\*”;

}

}

}

}

**Runtime**: θ(n2 lg n)

|  |  |
| --- | --- |
| ? | 0x27f3 |
| a | 10 |

|  |  |
| --- | --- |
| ? | 0x27f3 |
| a | 10 |

|  |  |
| --- | --- |
| ? | 0x27f3 |
| a | 10🡪35 |

|  |  |
| --- | --- |
| ? | 0x27f3 |
| a | 35 |

🡨 address of a

creates another variable w/ same address (can be called something else)

**Counting sort**

θ(max + n) operations, n is the # of elements and max is the biggest value in array

θ(max) extra memory (unstable), θ(max+n) for stable

In general, θ(n)

**Long Multiplication:** θ(n2)

**Fast multiplication:** θ(nlg(3)) = θ(n1.585)

**Red-Black Trees**

height: θ(lg(n))

search: θ(lg(n))

rotation: θ(1)

fixing: O(lg(n)), tree height

delete: O(lg(n)), tree height

d

d

**Russian Peasant Multiplication**

θ(lg(min(m, n)), m and n are the values to be multipled

**Lexicographic Permute**

θ(n) for each character, θ(n\*n!) for whole algorithm

If duplicates, θ(n\*n! / # times each letter is duplicated)

After “bhigadgfecb” comes “bhigaebcefg”

**Binary Search Trees**

Runtime: θ(# nodes)

The cist if search, insert, and delete are bounded by the tree height which is O(n) in worst case.

Search: O(tree height)

findMin/findMax: O(tree height)

Insertion: O(tree height)

Delete: O(tree height)

add25 function ends, clean up stack

d

Output: 35

x(n) = 2x(n/2) + n, x(1) = 1

Step 1: Step 2:

x(n/2) = 2x(n/4) + (n/2) x(n/4) = 2x(n/8) + (n/4)

x(n) = 2x(n/2) + n x(n) = 4x(n/4) + 2n

x(n) = 2[2x(n/4) + (n/2)] + n x(n) = 4[2x(n/8) + (n/4)] + 2n

x(n) = 4x(n/4) + 2(n/2) + n x(n) = 8x(n/8) + 4(n/4) + 2n

x(n) = 4x(n/4) + 2n x(n) = 8x(n/8) + 3n

Step 3: x(n) = 2kx(n/2k) + kn

Step 4: want x(1) so let n = 2k 🡪 k = lg(n)

Step 5:

x(n) = 2lg(n) \* x(2k/2k) + lg(n) \* n

x(n) = n \* x(1) + n lg(n) 🡪 x(n) = n lg(n) + n

The closer it is to x!, the worse the algorithm performs

Logarithmic 🡪 polynomial 🡪

exponential 🡪 factorial

void func1(int n) {

for(int i=n; i≥1; i--){

for(int j=n; j≥1; j-=1){

cout ≪ “\*”;

}

}

}

**Runtime: O(n2)**

**Divide and Conquer**

Example: x(n) = x(n/3) + n ∀n > 1, x(1) = 1, n = 3k

Step 1: Substitute n with 3k

x(3k) = x(3k/3) + 3k

x(3k) = x(3k-1) + 3k

Step 2: General form

x(3k) = x(3k-i) + k\*3k

Step 3: initial condition

k – i = 0 🡪 i = k

x(3k) = x(1) + k\*3k

x(3k) = 1 + k\*3k

Step 4: Solve

Since n = 3k, we can express k = log3(n):

x(n) = 1 + log3 n \* n

Step 1: Replace n with n/2

Step 2: Replace n with n/4

Step 3: General form

Step 4: initial condition

Step 5: Solve

**Replace and Conquer**

x(n) = x(n-1) + n, x(0) = 0

Step 1: Step 2:

x(n-1) = x(n-2) + n-1 x(n-2) = x(n-3) + n-2

x(n) = x(n-1) + n x(n) = x(n-2) + (n-1) + n

x(n) = x(n-2) + (n-1) + n x(n) = x(n-3) + (n-2) + (n-1) + n

Step 3:

x(n) = x(n-i) + x(n-i+1) +...+ n

Step 4: x(0) = 0 so n – i = o 🡪 i = n

Step 5: x(n) = 0+1+2+3+…+n = n(n+1)/2

void func4(int array[], int n) {

int sum = 0;

for(int i = 0; i < n; i++) {

if(array[i] < 0) {

return 0;

}

}

for(int i = 0; i < n; i++) {

sum += array[i];

}

return sum;

}

**Runtime**: O(n)

Is n2 + 2n + 6 ∈ O(n3)? YES

n2 + 2n ≤ n3 so c = 1

To find n0, we can either solve f(n)

|  |  |  |
| --- | --- | --- |
| n | n2 + 2n + 6 | n3 |
| 1 | 9 | 1 |
| 2 | 14 | 8 |
| 3 | 21 | 27 |

Since n = 3 is the first place where g(n) > f(n), n0 = 3.

Note on asymptotic notation:

∀ n ≥ n0, c1g(n)≤f(n)≤c2g(n) 🡪 g(n) is **asymptotically non-negative**, which means f(n) is non-negative when n is sufficiently large.

**Log Props**

loga(1) = 0

loga(a) = 1

loga(xy) = loga(x) + loga(y)

loga(x/y) = loga(x) - loga(y)

alog\_b(x) = xlog\_b(a)

loga(x) = logb(x)/logb(a)

func5 takes as input a sorted array of integers and second array of integer keys, applies binary search to every key in the second array, and returns an array of bool as to whether or not every key is in the sorted array. Both arrays have n elements. **Runtime: O(nlgn)**

int func2(int arr1[], arr2[], int n) {

int count = 0;

for(int i = 1; i ≤ n / i && arr[i] != arr[2]; i++) {

count++;

}

return count;

}

**Runtime: O(√n)**

**Asymptotic efficiency classes in increasing order of magnitude**

O(1), O(lg n), O(n), O(n lg n), O(n2), O(n2 lg n), O(n3), O(2n), O(n!), O(nn)

**Binary Gray Codes (length 3)**

[‘000’, ‘001’, ‘011’, ‘010’, ‘110’, ‘111’, ‘101’, ‘100’]

**Kahn’s Algorithm**

Uses array for in-degrees, list for result, and queue to track zero in-degrees during sorting

Pseudocode:

L 🡨 empty list that will contain the sorted elements

S 🡨 set of all nodes with no incoming edges

**while** S is non-empty **do**

remove a node n from S

add n to tail of L

**for each** node m with an edge e from n to m **do**

remove edge e from the graph (or decrease indegree)

**if** m has no other incoming edges **then**

insert m into S

**if** graph has edges **then**

return error (graph has a cycle)

**else**

return L (topologically sorted order)

Example:

A number grid with black numbers

Description automatically generated

Adjacency Matrix

A grid with numbers and a couple of black squares

Description automatically generated with medium confidence

**Graph Algorithms**

A diagram of a network

Description automatically generated

|  |  |  |
| --- | --- | --- |
|  | Matrix | List |
| Determining if there is an edge between 2 vertices | θ(1) | O(d)  where d is the degree if the vertex(# of edges)  O(V) |
| Determining all vertices adjacent to a given vertex | θ(V) | θ(d)  O(V) |
| Space requirement | θ(V2) | θ(V+E) |
| When to use | small and dense graphs | large and sparse graphs |

**BFS and DFS**

BFS: A, F, C, B, H, G, E, D

DFS: H, C, A, B, D, E, G, F

BFS visits vertex closest to Z

if there are multiple adjacent

DFS does opposite

BFS: root 🡪 neighbors

DFS: longest path then branches

**Master Theorem**

A math equations on a white background

Description automatically generated

Solve T(n) = 4T(n/2) + θ(n2)

a = 4, b = 2, d = 2

T(n) = θ(n2 lg n)

34, 12, 4, 56, 14, 2

34, 12, 4, 56, 2, 14

34, 12, 4, 2, 14, 56

34, 12, 2, 4, 14, 56

34, 2, 4, 12, 14, 56

2, 4, 12, 14, 34, 56

**Hoare Partition**

int hoare\_partition(int array[], int left, int right) {

int p = array[left], i=left-1, j=right+1;

while(true) {

i++;

while(array[i] < p) {

i++;

}

j--;

while(array[j] > p) {

j--;

}

if(i ≥ j) {

return j;

}

swap(array, i, j);

}

return j;

}

void quicksort(int array[], int left, int right) {

if (left < right) {

int s = hoare\_partition(array, left, right);

quicksort(array, left, s);

quicksort(array, s+1, right);

}

}

**Lomuto Partition**

// Partitions subarray by Lomuto’s algorithm using first element as pivot.

// Input: A subarray A[l..r] of array A[0..n−1], defined by its left and

// right indices l and r (l ≤ r)

// Output: Partition of A[l..r] and the new position s of the pivot.

LomutoPartition(A, l, r):

p ← A[l]

s ← l

for i ← l + 1 to r do

if A[i] < p

s ← s + 1

swap(A[s], A[i])

swap(A[l], A[s])

return s

Quicksort(A, l, r):

if(l < r)

s 🡨 LomutoPartition(A,l,r)

Quicksort(A, l, s-1)

Quicksort(A, s+1, r)

Mergesort(A, B, lo, hi)

if(lo < hi)

mid = lo + (hi-lo)/2

Mergesort(A, B, lo, mid)

Mergesort(A, B, mid+1, hi)

Merge(A, B, lo, mid, hi)

Merge(A, B, lo, mid, hi)

i1 = lo, i2 = mid, i=lo

while i1 ≤ mid and i2 ≤ hi

if A[i] ≤ A[i2]

B[i++] = A[i1++]

else

B[i++] = A[i2++]

for i1 to mid

B[i++] = A[i1++]

for i2 to hi

B[i++] = A[i2++]

copy B[lo...hi] back

into A[lo...hi]

**Horner’s Method**

Horner(P[0..n], x):

p = P[n] // p = aₙ

for i = n-1 downto 0:

p = P[i] + x \* p // p = aᵢ + x \* p

return p

**Left to Right Exponentiation:** useful for computing an.

First step: convert n into binary of the form n = bibi-1…b1b0

Pseudocode:

LeftRightBinaryExponentiation(a, b(n))

product <- a

for i <- (i-1) downto 0 do

product <- product \* product

if b\_i = 1:

product <- product \* a

return product

**Sorting Algorithms**

Bubble sort – compares adjacent elements in the array ad swapping them if they are out of order. Largest element “bubbles up to the end of the array”

void bubble\_sort(int array[], const int length) {

for(int i = 0; i < length - 1; i++) {

for(int j = 0; j < length - 1 - i; j++) {

if(array[j + 1] < array[j]) {

swap(array, j, j + 1);

}

}

}

}

**Example of bubble sort** 89 45 68 90 29 34 17

i = 0, j = 0 45-89 68 90 29 34 17 |

j = 1 45 68-89 90 29 34 17 |

j = 2 45 68 89 90 29 34 17 |

j = 3 45 68 89 29-90 34 17 |

j = 4 45 68 89 29 34-90 17 |

j = 5 45 68 89 29 34 17-90 | 6 comparisons, 5 swaps

i = 1, j = 0 45 68 89 29 34 17 | 90

j = 1 45 68 89 29 34 17 | 90

j = 2 45 68 29-89 34 17 | 90

j = 3 45 68 29 34-89 17 | 90

j = 4 45 68 29 34 17-89 | 90 5 comparisons, 3 swaps

i = 2, j = 0 45 68 29 34 17 | 89 90

j = 1 45 29-68 34 17 | 89 90

j = 2 45 29 34-68 17 | 89 90

j = 3 45 29 34 17-68 | 89 90 4 comparisons, 3 swaps

i = 3, j = 0 29-45 34 17 | 68 89 90

j = 1 29 34-45 17 | 68 89 90

j = 0 29 34 17-45 | 68 89 90 3 comparisons, 3 swaps

i = 4, j = 0 29 34 17 | 45 68 89 90

j = 1 29 17-34 | 45 68 89 90 2 comparisons, 1 swap

i = 5, j = 0 17-29 | 34 45 68 89 90 1 comparison, 1 swap

**Optimized Bubble Sort**

void bubble\_sort\_opt(int array[], const int length) {

int unsorted = length;

while(unsorted > 1) {

int s = 0;

for(int j = 1; j < unsorted; j++) {

if(array[j] < array[j - 1]) {

swap(array, j - 1, j);

s = j;

}

}

unsorted = s;

}

}

**Selection Sort** - finds the smallest element in the

array and swaps it with the element at the front of the array

void selection\_sort(int array[], const int length) {

for(int i = 0; i < length - 1; i++) {

int min\_j = i;

for(int j = i + 1; j < length; j++) {

if(array[j] < array[min\_j]) {

min\_j = j;

}

}

if(min\_j != i) {

swap(array, i, min\_j);

}

}

}

**Example 89 45 68 90 29 34 17**

i = 0, j = 1 | 89 45<68 90 29 34 17

j = 2 | 89 45<68 90 29 34 17

j = 3 | 89 45<68 90 29 34 17

j = 4 | 89 45 68 90 29<34 17

j = 5 | 89 45 68 90 29<34 17

j = 6 | 89 45 68 90 29 34 17<

swap | 17 45 68 90 29 34 89 6 comparisons, 1 swap

i = 1, j = 2 17 | 45<68 90 29 34 89

j = 3 17 | 45<68 90 29 34 89

j = 4 17 | 45 68 90 29<34 89

j = 5 17 | 45 68 90 29<34 89

j = 6 17 | 45 68 90 29<34 89

swap 17 | 29 68 90 45 34 89 5 comparisons, 1 swap

i = 2, j = 3 17 29 | 68<90 45 34 89

j = 4 17 29 | 68 90 45<34 89

j = 5 17 29 | 68 90 45 34<89

j = 6 17 29 | 68 90 45 34<89

swap 17 29 | 34 90 45 68 89 4 comparisons, 1 swap

i = 3, j = 4 17 29 34 | 90 45<68 89

j = 5 17 29 34 | 90 45<68 89

j = 6 17 29 34 | 90 45<68 89

swap 17 29 34 | 45 90 68 89 3 comparisons, 1 swap

i = 4, j = 5 17 29 34 45 | 90 68<89

j = 6 17 29 34 45 | 90 68<89

swap 17 29 34 45 | 68 90 89 2 comparisons, 1 swap

i = 5, j = 6 17 29 34 45 68 | 90 89<

swap 17 29 34 45 68 | 89 90 1 comparison, 1 swap

**Insertion Sort**: takes the first element in the unsorted part of the array and inserts it in the correct place in the array that has already been sorted

void insertion\_sort(int array[], const int length) {

for(int i = 1; i < length; i++) {

int j, current = array[i];

for(j = i - 1; j >= 0 && array[j] > current; j--) {

array[j + 1] = array[j];

}

array[j + 1] = current;

}

}

Example: 89 | 45 68 90 29 34 17

current = 45

i = 1, j = 0 89>| 89 68 90 29 34 17

insert 45 89 | 68 90 29 34 17 1 comparison, 1 shift

current = 68

i = 2, j = 1 45 89>| 89 90 29 34 17

j = 0 45 89 | 89 90 29 34 17

insert 45 68 89 | 90 29 34 17 2 comparisons, 1 shift

current = 90

i = 3, j = 2 45 68 89 | 90 29 34 17

insert 45 68 89 90 | 29 34 17 1 comparison, 0 shift

current = 29

i = 4, j = 3 45 68 89 90>| 90 34 17

j = 2 45 68 89>89 | 90 34 17

j = 1 45 68>68 89 | 90 34 17

j = 0 45>45 68 89 | 90 34 17

insert 29 45 68 89 90 | 34 17 4 comparisons, 4 shifts

current = 34

i = 5, j = 4 29 45 68 89 90>| 90 17

j = 3 29 45 68 89>89 | 90 17

j = 2 29 45 68>68 89 | 90 17

j = 1 29 45>45 68 89 | 90 17

j = 0 29 45 45 68 89 | 90 17

insert 29 34 45 68 89 90 | 17 5 comparisons, 4 shifts

current = 17

i = 6, j = 5 29 34 45 68 89 90>| 90

j = 4 29 34 45 68 89>89 | 90

j = 3 29 34 45 68>68 89 | 90

j = 2 29 34 45>45 68 89 | 90

j = 1 29 34>34 45 68 89 | 90

j = 0 29>29 34 45 68 89 | 90

insert 17 29 34 45 68 89 90 | 6 comparisons, 6 shifts

A number in a rectangular box

Description automatically generated with medium confidence

**Binary Reflected Gray Codes**

def BRGC(n):

if n == 1:

return [‘0’, ‘1’]

L1 = BRGC(n-1)

L2 = L1[∷-1]

L3 = [‘0’ + code for code in L1]

L4 = [‘1’ + code for code in L2]

return L3+L4

**Binary Search Tree Terms**

**length**: # edges along a path

**depth**: length of a unique path to

root

**height** **of node**: length of longest

path to a leaf

**height of tree**: depth of the deepest

leaf, lone root = 0, empty tree = -1

**size**: # nodes in tree

**max width of tree**: max # of nodes having

the same depth

**diameter of tree:** length of longest path

between any 2 nodes

**Traversing**

Preorder: root, left, right

Inorder: left, root, right

Postorder: left, right, root

**Deletion**

Case 1: leaf node 🡪 just delete it

Case 2: has 1 child 🡪 point to next child

A diagram of a number tree

Description automatically generated with medium confidence

Case 3: replace with minimum node in right subtree or maximum node in left subtree

A diagram of a tree

Description automatically generated

**Candies**

Alice is a kindergarten teacher. She wants to give some candies to the children in her class. All the children sit in a line and each of them has a rating score according to his or her performance in the class. Alice wants to give at least **TWO** candies to each child. If two children sit next to each other, then the one with the higher rating must get more candies (if the ratings are the same then the children can get the same number of candies or a different number). Alice wants to minimize the total number of candies she must buy.

Performance 2 1 5 4 1 7 8 3 1

Candies a1 a2 a3 a4 a5

a1 = 3, a2 = 4, a3 = 3, a4 = 4, a5 = 3

Fill in the blanks

# Make a forward pass, looking for children who should receive

# more candies than the one to their left.

for i in range(n-1):

if P[ i+1 ] > P[ i ]:

C[ i+1 ] = **C[i] + 1**

# Make a reverse pass, looking for children who

should receive

# more candies than the one to their right.

for i in range(n-1, 0, -1):

if P[ i-1 ] > P[ i ] and C[ i-1 ] <= C[ i ]:

**C[i-1]** = **C[i]+1**

- makeset(x) creates a tree containing only the

vertex x.

Running time: Θ(1), so Θ(V) for all vertices together.

- union(x, y) attaches the root of the tree containing

y to the root of the tree containing x (the root of the

tree containing x becomes the parent

of the root of the tree containing y).

Running time: Θ(1).

- find(x) returns the root of tree containing x

(by following the parent pointers from x all

the way to the root).

Running time: O(V)

Prim’s & Kruskal’s

A diagram of a tree

Description automatically generatedA table of equations with circles and lines

Description automatically generated

**Russian Peasant Multiplication**

1. n < m

2. n/2 and m\*2

3. Keep the m values when n is odd

4. Add all m values

Example: n = 238, m = 13

Swap. n = 13, m = 238

|  |  |  |
| --- | --- | --- |
| n | m | n\_odd |
| 13 | 238 | 238 |
| 6 | 476 |  |
| 3 | 952 | 952 |
| 1 | 1904 | 1904 |
| Answer: | | 3094 |

Adjacency Matrix: y axis: from, x-axis: to

Adjacency list: 1 goes to 2 and 4

If you have a directed graph with 20 edges and 380 vertices, use list

If there is an edge from every vertex to every vertex including itself, the relation between V and E is E=((V\*(V-1))/2)+V because V\*(V-1)/2 is for the distinct vertices and the +V is for the edges to itself

**RBT Rules**

Properties:

1. Each node is red or black 2. Root must be black 3. Every leaf (represented as nulls) are black 4. If a node is red, both its children are black

5. All paths from a node to its leaves contain the same number of black nodes

6. A RBT with n internal nodes has height at most 2 \* lg(n+1)

Insertion into RBT

2. The root is black 4. If a node is red, then both its children are black

**Counting Sort**

void **counting\_sort**(int array[], const int length) {

int max = 0;

for(int i = 0; i < length; i++) {

if(array[i] > max) {

max = array[i];

}

}

int \*count = new int[max + 1];

fill(count, count + max + 1, 0);

for(int i = 0; i < length; i++) {

count[array[i]]++;

}

for(int i = 0, j = 0; i <= max; i++) {

while(count[i]--) {

array[j++] = i;

}

}

delete[] count;

}

Horner’s Method

Given P(x) = 2x4-x3+3x2+x-5, the array will be [-5, 1, 3, 01, 2]. Evaluate x0 = 3

x p n i

3 2 4

5 3 p=3\*2-1

18 2 p=3\*5+3

55 1 p=3\*18+1

**160** 0 p=3\*55-5

**Warshall-Floyd’s**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** |
| **1** | 0 | 3 | ∞ | 7 |
| **2** | 8 | 0 | 2 | ∞ |
| **3** | 5 | ∞ | 0 | 1 |
| **4** | 2 | ∞ | ∞ | 0 |

k=1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** |
| **1** | 0 | 3 | ∞ | 7 |
| **2** | 8🡪5 | 0 | 2 | ∞ |
| **3** | 5 | ∞ | 0 | 1 |
| **4** | 2 | ∞ | ∞ | 0 |

There is an edge going 1 🡪 3

y axis is from, x is to

If there are n children in the class, what is the asymptotic running time to determine how many candies each child should receive? **THETA(n)**

Starting array: [7,8,9,4,2,7]

7 is pivot & 7<8,9 and 7>4,2,7 so no changes are made

Starting array: [6,2,8,4,3,9]

With 6 and 2, nothing. With 8.. [6,2,9,4,3,8]

Starting array: [7,6,13,12,0,7]

One time with Lomuto: [6,0,7,12,13,7]

If after 1 Lomuto, the array is [4,3,5,6,8,7], the pivot=6

**Lomuto**: compare element pivot+1, with pivot. If element > pivot, increment i and swap, if not, check next element

Example: calculate 47

a = 4, n = 8, b(n) = 111 <- 7

product a i bi

4 4 1 1

42\*42 = 64 0 1

642 = **16384**

A diagram of numbers and circles

Description automatically generated

Preorder: root, left, right – print when touch L side

[8, 3, 1, 6, 4, 7, 10, 14, 13]

Inorder: left, root, right – print when touch bottom

[1, 3, 4, 6, 7, 8, 10, 13, 14]

Postorder: left, right, root – print when touch R side

[1, 4, 7, 6, 3, 13, 14, 10, 8]

**Lexicographic Permute**

1. Find longest suffix without increases

2. The character right before suffix is the pivot

3. Find the character greater than the pivot in the suffix

4. Swap pivot and suffix character

5. Reverse suffix

6. Concatenate prefix, swapped pivot, and reversed suffix

LexicographicPermute(n)

// Generates permutations in lexicographic order

// Input: positive integer n

// Output: list of all permutations of {1...n} in lexicographic order

**while** last permutation has 2 consecutive elements in increasing order **do**

**let** i = largest index | ai < ai+1

find largest index j | ai < aj

swap ai with aj

reverse the order of the elements from ai+1 to an inclusive

add the new permutation to the list

void some\_sort(int array[], const int length) {

bool sorted = false;

while(!sorted) {

sorted = true;

for(int i=1; i<length-1; i+=2) {

if(array[i] > array[i+1]) {

swap(array, i, i+1);

sorted = false;

}

}

for(int i=0;i < length-1; i+=2) {

if(array[i] > array[i+1]) {

swap(array, i, i+1);

sorted = false;

}

}

}

}

Example: lowest value first



1, 2, 5, 4, 3, 7, 6

1 2 3 4 5 6 7

0 1 2 3 1 3 2

0 1 2 0 2 1

0 1 1 0

0 0

Quickselect

// Solves the selection problem by recursive partition-based algorithm

// Input: Subarray A[l...r] of array A[0...n-1] of orderable elements and integer k (k ≤ n-1)

// Output: value of kth smallest element

Quickselect(A, l, r, k):

s 🡨 LomutoPartition(A, l, r)

if s = k-1 return A[s]

else if s > k-1 Quickselect(A, l, s-1, k)

else Quickselect(A, s+1, r, k)

best case: θ(n)

worst case: θ(n2)

resembles bubble sort

Selection

sort

F

A

C

H

G

B

E

D