CS32 Discussion
Week 7

Muhao Chen
muhaochen@ucla.edu
http://yellowstone.cs.ucla.edu/~muhao/
Outline

• Big-O Notation
• Sorting
• Tree
Complexity

• Quantify the efficiency of a program.
• Magnitude of the time and space cost for an algorithm given certain size of input.
  • Time complexity: quantifies the run time.
  • Space complexity: quantifies the usage of the memory (or sometimes hard disk drives, cloud disk drives, etc.).
Size of input vs. Running time

• A program gets some kind of input, does something meaningful (hopefully), and produces some output.

• Naturally, the size of input determines how long a program runs.
  – Sorting an array of 1000 items should run a lot longer than sorting an array of 10 items.
  – But how much longer?

• Sometimes, the size of input doesn’t matter.
  – Figuring out the size of a C++ string (s.size()): always the same running time.
Big Question

- Given an input of size \( n \), approximately how long does the algorithm take to finish the task, in terms of \( n \)?
  
  \( \rightarrow \) Big-O notation
Formal Definition of Big-O

• Let $T(n)$ be the function that measures the runtime of the program given $n$ size of input.

• Let $g(n)$ be another function defined on the real number field.

  • $T(n) = O(g(n))$ iff $\exists M > 0$ and $\exists N > 0$ s.t. $\forall n > N : T(n) \leq M \times g(n)$

When the input size $n$ grows above certain scale $N$, the runtime $T(n)$ is of the same or less magnitude of the function $g$ inside $O(\ )$. 

Upper bound
Big-O

- If your algorithm takes …
  - about n steps: $O(n)$
  - about 2n steps: $O(n)$
  - about $n^2$ steps: $O(n^2)$
  - about $3n^2 + n$ steps: $O(n^2)$
  - about $2^n$ steps: $O(2^n)$

- Which one grows faster in the long term?
  - 10000n vs. 0.00001$n^2$
Big-O

![Graph showing the growth of different order functions: O(2^n), O(n^3), O(n^2), O(n), O(n) compared to running time as a function of n.](image-url)
Efficiency

• Algorithms are considered efficient if it runs reasonably fast even with a large input.
• Algorithms are considered inefficient if it runs slow even with a small input.

• For more precise definitions, take CS180 and 181. In this class, we will use these simple intuitions to analyze algorithms.
Linear Search

- Unsorted array – look for an item

```c
linear_search( array arr, size n, value v )
{
    for (i=0 to n-1)
    {
        if (arr[i] == v)          Best case?
            return i;
    }
    return -1;                 Worst case?
}
```
Linear Search

- Unsorted array – look for an item

```cpp
linear_search( array arr, size n, value v )
{
    for (i=0 to n-1)
    {
        if (arr[i] == v)
            return i;
    }
    return -1;
}
```

Best case?
- \( v \) is found in the first slot (\( a[0] \)) – takes 1 step

Average case?
- takes \( n/2 = \frac{1}{2}n \) steps (assuming \( v \) can appear at any location in the array with an equal probability)

Worst case?
- not found – \( n \) steps
Linear Search

- **Unsorted array – look for an item**

```java
linear_search( array arr, size n, value v )
{
    for (i=0 to n-1)
    {
        if (arr[i] == v)
            return i;
    }
    return -1;
}
```

- **Best case?**
  - `v` is found in the first slot (`a[0]`) – $O(1)$

- **Average case?**
  - $O(n)$ (assuming `v` can appear at any location in the array with an equal probability)

- **Worst case?**
  - not found – $O(n)$
All Pairs

- Find all ordered pairs

```java
all_pairs( array arr, size n )
{
    for (i=0 to n-1)
        for (j=0 to n-1)
            if (i ≠ j)
                print "{arr[i] arr[j]}";
}
```

Best case?
Average case?
Worst case?
All Pairs

- Find all ordered pairs

```c
all_pairs( array arr, size n )
{
    for (i=0 to n-1)
        for (j=0 to n-1)
            if (i ≠ j)
                print "{arr[i] arr[j]}";
}
```

Best case?
Average case?
Worst case?

All $O(n^2)$
Unit Operations (O(1) operations)

- Addition, Subtraction, Multiplication, Division
- Comparison, Assignment
- Input, Output of a small value (e.g. short string, an integer, etc.)

- If $O(1)$ operations are repeatedly done in a loop for $n$ times, then that loop is $O(n)$.
- If this loop is within a loop that repeats $n$ times, then this outer loop takes $O(n^2)$. 
Big-O Arithmetic

- More generally:
- If things happen sequentially, we add Big-Os.
  - $O(1)$ operation followed by $O(1) = O(1) + O(1) = O(1)$
- If one thing happens within another, then we multiply Big-Os.
  - $O(1)$ operation within a $O(n)$ loop = $O(1) \times O(n) = O(n)$

- $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$
- $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$
Linear Search

• Unsorted array – look for an item

```c
linear_search( array arr, size n, value v )
{
    for (i=0 to n-1) O(n)
    {
        if (arr[i] == v) O(1)
            return i;
    }
    return -1;
}
```

Best case?
- v is found in the first slot (a[0]) – O(1)

Average case?
- O(n) (assuming v can appear at any location in the array with an equal probability)

Worst case?
- not found – O(n)

Usually, assessing complexity involves counting nested loops – only one loop means it’s likely to be O(n)
All Pairs

- Find all ordered pairs

```c
all_pairs( array arr, size n )
{
    for (i=0 to n-1)  O(n)
        for (j=0 to n-1)  O(n)
            if (i ≠ j)
                print "{arr[i] arr[j]}");

    O(1)  

    Best case?
    Average case?
    Worst case?

    All O(n²)
```
## Order of Complexity

<table>
<thead>
<tr>
<th>Big O</th>
<th>Name</th>
<th>n = 128</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(1))</td>
<td>constant</td>
<td>1</td>
</tr>
<tr>
<td>(O(\log n))</td>
<td>logarithmic</td>
<td>7</td>
</tr>
<tr>
<td>(O(n))</td>
<td>linear</td>
<td>128</td>
</tr>
<tr>
<td>(O(n \log n))</td>
<td>“n log n”</td>
<td>896</td>
</tr>
<tr>
<td>(O(n^2))</td>
<td>quadratic</td>
<td>16192</td>
</tr>
<tr>
<td>(O(n^k), k \geq 1)</td>
<td>polynomial</td>
<td></td>
</tr>
<tr>
<td>(O(2^n))</td>
<td>exponential</td>
<td>(10^{40})</td>
</tr>
<tr>
<td>(O(n!))</td>
<td>factorial</td>
<td>(10^{214})</td>
</tr>
</tbody>
</table>
Binary Search

• Find an item $v$ in a *sorted* array

```cpp
binary_search( array arr, value v, start index s, end index e )
{
    if (s > e)
        return -1
    find the middle point $i = (s + e) / 2$
    if (arr[i] == v)
        return i
    else if (arr[i] < v)
        return binary_search(arr, v, i+1, e)
    else
        return binary_search(arr, v, s, i-1)
}
```
Binary Search

- **Find an item v in a sorted array**

```c
binary_search( array arr, value v, start index s, end index e )
{
    if (s > e)
        return -1;
    find the middle point i = (s + e) / 2
    if (arr[i] == v)
        return i;
    else if (arr[i] < v)
        return binary_search(arr, v, i+1, e);
    else
        return binary_search(arr, v, s, i-1);
}
```

**Best case?**
O(1) – by now you can see the best case analysis doesn’t help much

**Average case?**
O(log n)

**Worst case?**
O(log n)
Binary Search

• At every iteration, we divide the search space in half.
• You keep dividing the size by 2 until it becomes 1.
• It takes $\sim \log_2 n$ steps to get to 1.
• $\log_{10} n = (1 / \log_2 10) \log_2 n$
• So the base does not matter.
Why Big-O’s are important

• You’ll be asked about it in job interviews!!!!!
Sorting Algorithms

• We now switch gears and discuss some well known sorting algorithms.
Selection Sort

- Find the smallest item in the unsorted portion, and place it in front.

- What is the running time (complexity) of this algorithm?
Insertion Sort

- Pick one from the unsorted part, and place it in the “right” position in the sorted part.
- Best case?
- Avg. case?
- Worst case?
Insertion Sort

- Pick one from the unsorted part, and place it in the “right” position in the sorted part.
- Best case? $O(n)$
- Avg. case? $O(n^2)$
- Worst case? $O(n^2)$
Merge Sort

3 7 6 5

3 7 5 6

8 2 1 4

2 8 1 4

3 5 6 7

1 2 4 8

1 2 3 4 5 6 7 8

Merge
Merge Sort: Running Time?

3 7 6 5

3 7 5 6

3 5 6 7

1 2 3 4 5 6 7 8

O(n)

O(n)

O(n)

O(n)

O(log n)

O(n)O(log n) = O(n log n)
General Sorting: Running Time

- $O(n \log n)$ is faster than $O(n^2)$ – merge sort is more efficient than selection sort or insertion sort.
- $O(n \log n)$ is the best average complexity that a general (comparison) sorting algorithm can get (assuming you know nothing about the data set).
- With more information about the data set provided, you can sometimes sort things almost linearly.
Quick Sort

- Pick a **pivot**, and move numbers that are less than the pivot to front, and ones that are greater than the pivot to end. (Does this sound familiar?)
- On average, $O(n \log n)$
- Depending on how you pick your pivots, it can be as bad as $O(n^2)$
void permutation(vector<int>& nums, int start) {
    if (start == nums.size() - 1) {
        for(int i=0; i<nums.size(); ++i)
            cout << nums[i] << ' ,';
        cout << endl;
    }
    permutation(nums, start + 1);
    for (int i=start+1; i<nums.size(); ++i) {
        swap(nums[start], nums[i]);
        permutation(nums, start + 1);
        swap(nums[start], nums[i]);
    }
}
permutation(nums, 0); //call this function

O(n!)
Quick Questions

• Given an unsorted array of n items, what is the best you can do to search for an item, if you are to run this search only once?

• Given an unsorted array of n items, what is the best you can do to search for an item, if you are to run this search 100 times? (assume: n >> 100)

• Given an unsorted array of n items, what is the best you can do to search for an item, if you are to run this search n times?
Tree
Tree: Definitions

node → link (edge)

parent

children

siblings

leaves

root

A

B

C

D

E

F

G

H
Tree: Definitions

- **Node**
- **Link (edge)**
- **Parent**
- **Children**
- **Siblings**
- **Leaves**
- **Height**
- **Subtree**

No loop!
Bound on # of edges

How many edges should there be in a tree of $n$ nodes?

$n - 1$
Binary Trees

No node has more than 2 children (left child + right child).
Binary Trees

How many nodes can a binary tree of height $h$ have? (one with max. # of nodes == full binary tree)

$2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$
Tree is a data structure!

- For every data structure we need to know:
  - how to **insert** a node,
  - how to **remove** a node,
  - **search** for a node

- and (for tree only)
  - how to traverse the tree

```c
struct Node {
    ItemType val;
    Node* left;
    Node* right;
};
```
Three Methods of Traversal

```cpp
void preorder(const Node *node) {
    if (node == NULL) return;
    cout << node->val << " ";
    preorder(node->left);
    preorder(node->right);
}

void inorder(const Node *node) {
    if (node == NULL) return;
    inorder(node->left);
    cout << node->val << " ";
    inorder(node->right);
}

void postorder(const Node *node) {
    if (node == NULL) return;
    postorder(node->left);
    postorder(node->right);
    cout << node->val << " ";
}
```
int treeHeight(const Node *node) {
    if (node == NULL)
        return 0;

    int leftHeight = treeHeight(node->left);
    int rightHeight = treeHeight(node->right);

    if (leftHeight > rightHeight)
        return leftHeight + 1;
    else
        return rightHeight + 1;
}
Bugs in your software are actually special features :)

```php
if ($thirsty == TRUE) {
} else {
}
```