CMPT 225

Algorithm Analysis
Algorithm Analysis

- Measuring algorithm efficiency
  - Timing
  - Counting
- Cost functions
  - Cases
    - Best case
    - Average case
    - Worst case
  - Searching
  - Sorting
- O Notation
  - O notation's mathematical basis
  - O notation classes
  - \( \Theta \) and \( \Omega \) notations
Counting
Algorithms can be described in terms of time and space efficiency.

Time
- How long, in ms, does my algorithm take to solve a particular problem?
- How much does the time increase as the problem size increases?
- How does my algorithm compare to other algorithms?

Space
- How much memory space does my algorithm require to solve the problem?
Choosing an appropriate algorithm can make a significant difference in the usability of a system

- Government and corporate databases with many millions of records, which are accessed frequently
- Online search engines and media platforms
- Big data
- Real time systems where near instantaneous response is required
  - From air traffic control systems to computer games
There are often many ways to solve a problem
- Different algorithms that produce the same results
  - e.g. there are numerous *sorting* algorithms
- We are usually interested in how an algorithm performs when its input is large
  - In practice, with today's hardware, *most* algorithms will perform well with small input
  - There are exceptions to this, such as the *Traveling Salesman Problem*
    - Or the recursive Fibonacci algorithm presented previously ...
It is possible to count the number of operations that an algorithm performs
- By a careful visual walkthrough of the algorithm or by
- Inserting code in the algorithm to count and print the number of times that each line executes

It is also possible to time algorithms
- Compare system time before and after running an algorithm
  - More sophisticated timer classes exist
  - Simply timing an algorithm may ignore a variety of issues
Timing Algorithms

- It may be useful to time how long an algorithm takes to run
  - In some cases it may be *essential* to know how long an algorithm takes on a particular system
    - e.g. air traffic control systems
    - Running time may be a strict requirement for an application
- But is this a good *general* comparison method?
  - Running time is affected by a number of factors other than algorithm efficiency
Running Time is Affected By

- CPU speed
- Amount of main memory
- Specialized hardware (e.g. graphics card)
- Operating system
- System configuration (e.g. virtual memory)
- Programming language
- Algorithm implementation
- Other programs
- System tasks (e.g. memory management)
- ...

John Edgar
Instead of *timing* an algorithm, *count* the number of instructions that it performs.

The number of instructions performed may vary based on:
- The size of the input
- The organization of the input

The number of instructions can be written as a cost function on the input size.
void printArray(int arr[], int n){
    for (int i = 0; i < n; ++i){
        cout << arr[i] << endl;
    }
}

Operations performed on an array of length 10
- declare and initialize i
- perform comparison, print array element, and increment i: 10 times
- make comparison when i = 10

32 operations
Cost Functions

- Instead of choosing a particular input size we will express a cost function for input of size \( n \)
  - We assume that the running time, \( t \), of an algorithm is proportional to the number of operations
- Express \( t \) as a function of \( n \)
  - Where \( t \) is the time required to process the data using some algorithm \( A \)
  - Denote a cost function as \( t_A(n) \)
    - i.e. the running time of algorithm \( A \), with input size \( n \)
A Simple Example

void printArray(int arr[], int n){
    for (int i = 0; i < n; ++i){
        cout << arr[i] << endl;
    }
}

Operations performed on an array of length $n$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3n</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>declare and</td>
<td></td>
<td>perform comparison,</td>
<td>make comparison</td>
</tr>
<tr>
<td>initialize $i$</td>
<td></td>
<td>print array element, and</td>
<td>when $i = n$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>increment $i$: $n$ times</td>
<td></td>
</tr>
</tbody>
</table>

$t = 3n + 2$
In the example we assumed two things
- Neither of which are strictly true ...
- Any C++ statement counts as a single operation
  - Unless it is a function call
- That all operations take the same amount of time
  - Some fundamental operations are faster than others
  - What is a fundamental operation in a high level language is multiple operations in assembly
- These are both simplifying assumptions
The number of operations often varies based on the size of the input

- Though not always – consider array lookup

- In addition algorithm performance may vary based on the organization of the input
  - For example consider searching a large array
  - If the target is the first item in the array the search will be very fast
Best, Average and Worst Case

- Algorithm efficiency is often calculated for three broad cases of input
  - Best case
  - Average (or “usual”) case
  - Worst case
- This analysis considers how performance varies for different inputs of the same size
Analyzing Algorithms

- It can be difficult to determine the exact number of operations performed by an algorithm
  - Though it is often still useful to do so
- An alternative to counting all instructions is to focus on an algorithm's *barometer instruction*
  - The barometer instruction is the instruction that is executed the most number of times in an algorithm
  - The number of times that the barometer instruction is executed is usually proportional to its running time
Cost Functions for Searching
Searching

- It is often useful to find out whether or not a list contains a particular item
  - Such a search can either return true or false
  - Or the position of the item in the list
- If the array isn't sorted use *linear search*
  - Start with the first item, and go through the array comparing each item to the target
  - If the target item is found return true (or the index of the target element)
int linearSearch(int arr[], int n, int x){
    for (int i=0; i < n; i++){
        if(arr[i] == x){
            return i;
        }
    }
    //for
    return -1; //target not found
}

The function returns as soon as the target item is found.

return -1 to indicate that the item has not been found.

Worst case cost function: $t_{linear search} = 3n+2$
Linear Search Barometer Instruction

- Search an array of $n$ items
- The barometer instruction is equality checking (or *comparisons* for short)
  - `arr[i] == x;`
  - There are actually two other barometer instructions
    - What are they?
- How many comparisons does linear search perform?

```c
int linearSearch(int arr[], int n, int x){
    for (int i=0; i < n; i++){
        if(arr[i] == x){
            return i;
        }
    } //for
    return -1; //target not found
}
```

the *barometer operation* is the most frequently executed operation
Linear Search Comparisons

- **Best case**
  - The target is the first element of the array
  - Makes 1 comparison

- **Worst case**
  - The target is not in the array or
  - The target is at the last position in the array
  - Makes $n$ comparisons in either case

- **Average case**
  - Is it \((\text{best case} + \text{worst case}) / 2\), i.e. \((n + 1) / 2\)?
There are two situations when the worst case occurs

- When the target is the last item in the array
- When the target is not there at all

To calculate the average cost we need to know how often these two situations arise

- We can make assumptions about this
- Though these assumptions may not hold for a particular use of linear search
Assumptions

- **A1:** The target is not in the array half the time
  - Therefore half the time the entire array has to be checked to determine this

- **A2:** There is an equal probability of the target being at any array location
  - If it is in the array
  - That is, there is a probability of $1/n$ that the target is at some location $i$
Cost When Target Not Found

- Work done if the target is not in the array
  - $n$ comparisons
  - This occurs with probability of 0.5 (A1)
Cost When Target Is Found

- Work done if target is in the array:
  - 1 comparison if target is at the 1\textsuperscript{st} location
    - Occurs with probability $\frac{1}{n}$ (A2)
  - 2 comparisons if target is at the 2\textsuperscript{nd} location
    - Also occurs with probability $\frac{1}{n}$
  - $i$ comparisons if target is at the $i$\textsuperscript{th} location

- Take the weighted average of the values to find the total expected number of comparisons ($E$)
  - $E = 1\times\frac{1}{n} + 2\times\frac{1}{n} + 3\times\frac{1}{n} + \ldots + n\times\frac{1}{n}$ or
  - $E = \frac{(n + 1)}{2}$
Average Case Cost

- Target is *not* in the array: \( n \) comparisons
- Target *is* in the array \( (n + 1) / 2 \) comparisons
- Take a weighted average of the two amounts:
  - \( = (n \times \frac{1}{2}) + ((n + 1) / 2 \times \frac{1}{2}) \)
  - \( = (n / 2) + ((n + 1) / 4) \)
  - \( = (2n / 4) + ((n + 1) / 4) \)
  - \( = (3n + 1) / 4 \)
- Therefore, on average, we expect linear search to perform \( (3n + 1) / 4 \) comparisons
If we sort the target array first we can change the linear search average cost to approximately $n/2$
- Once a value equal to or greater than the target is found, the search can end
  - So, if a sequence contains 8 items, on average, linear search compares 4 of them,
  - If a sequence contains 1,000,000 items, linear search compares 500,000 of them, etc.
- However, if the array is sorted, it is possible to do much better than this by using binary search
int binarySearch(int arr[], int n, int x){
    int low = 0;
    int high = n - 1;
    int mid = 0;
    while (low <= high){
        mid = (low + high) / 2;
        if(x == arr[mid]){
            return mid;
        } else if(x > arr[mid]){  // Arranged if block
            low = mid + 1;
        } else { //x < arr[mid]
            high = mid - 1;
        }
    }
    return -1;  // Target not found
}
The algorithm consists of three parts
- Initialization (setting lower and upper)
- While loop including a return statement on success
- Return statement which executes on failure
- Initialization and return on failure require the same amount of work regardless of input size
- The number of times that the while loop iterates depends on the size of the input
The while loop contains an *if, else if, else* statement

The first if condition is met when the target is found
- And is therefore performed at most once each time the algorithm is run

The algorithm usually performs 5 operations for each iteration of the while loop
- Checking the while condition
- Assignment to mid
- Equality comparison with target
- Inequality comparison
- One other operation (setting either lower or upper)
Best Case

- In the best case the target is the midpoint element of the array
  - Requiring just one iteration of the while loop

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td>17</td>
<td>19</td>
<td>23</td>
</tr>
</tbody>
</table>

binary search (arr, 11)

mid = (0 + 7) / 2 = 3
What is the worst case for binary search?

- Either the target is not in the array, or
- It is found when the search space consists of one element

How many times does the while loop iterate in the worst case?

```
binary search (arr, 20)
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td>17</td>
<td>19</td>
<td>23</td>
</tr>
</tbody>
</table>

```
mid = (0 + 7) / 2 = 3  (4 + 7) / 2 = 5  (6 + 7) / 2 = 6  (7 + 7) / 2 = 7
```
Analyzing the Worst Case

- Each iteration of the while loop halves the search space
  - For simplicity assume that \( n \) is a power of 2
    - So \( n = 2^k \) (e.g. if \( n = 128 \), \( k = 7 \) or if \( n = 8 \), \( k = 3 \))

- How large is the search space?
  - After the first iteration the search space is \( n/2 \)
  - After the second iteration the search space is \( n/4 \)
  - After the \( k^{th} \) iteration the search space consists of just one element
    - Note that as \( n = 2^k \), \( k = \log_2 n \)
  - The search space of size 1 still needs to be checked
  - Therefore at most \( \log_2 n + 1 \) iterations of the while loop are made in the worst case

Cost function: \( \text{t}_{\text{binary search}} = 5(\log_2 (n)+1)+4 \)
Is the average case more like the best case or the worst case?

- What is the chance that an array element is the target?
  - \(\frac{1}{n}\) the first time through the loop
  - \(\frac{1}{(n/2)}\) the second time through the loop
  - ... and so on ...

- It is more likely that the target will be found as the search space becomes small
  - That is, when the while loop nears its final iteration
  - We can conclude that the average case is more like the worst case than the best case
## Binary Search vs Linear Search

<table>
<thead>
<tr>
<th>n</th>
<th>Linear Search $(3n+1)/4$</th>
<th>Binary Search $\log_2(n)+1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>76</td>
<td>8</td>
</tr>
<tr>
<td>1,000</td>
<td>751</td>
<td>11</td>
</tr>
<tr>
<td>10,000</td>
<td>7,501</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>75,001</td>
<td>18</td>
</tr>
<tr>
<td>1,000,000</td>
<td>750,001</td>
<td>21</td>
</tr>
<tr>
<td>10,000,000</td>
<td>7,500,001</td>
<td>25</td>
</tr>
</tbody>
</table>
Simple Sorting
Simple Sorting

- As an example of algorithm analysis let's look at two simple sorting algorithms
  - Selection Sort and
  - Insertion Sort
- Calculate an approximate cost function for these two sorting algorithms
  - By analyzing how many operations are performed by each algorithm
  - This will include an analysis of how many times the algorithms' loops iterate
Selection Sort

- The array is divided into sorted part and unsorted parts
- Expand the sorted part by swapping the first unsorted element with the smallest unsorted element
  - Starting with the element with index 0, and
  - Ending with the last but one element (index $n - 1$)
- Requires two processes
  - Finding the smallest element of a sub-array
  - Swapping two elements of the array

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>21</td>
<td>27</td>
<td>29</td>
<td>23</td>
<td>19</td>
<td>13</td>
<td>31</td>
<td>15</td>
<td>17</td>
<td>9</td>
<td>25</td>
<td>11</td>
</tr>
</tbody>
</table>

smallest: 9

The algorithm is on its fifth iteration
Find the smallest element in arr[4:15]
Selection Sort

- The array is divided into sorted part and unsorted parts
- Expand the sorted part by swapping the first unsorted element with the smallest unsorted element
  - Starting with the element with index 0, and
  - Ending with the last but one element (index $n - 1$)
- Requires two processes
  - Finding the smallest element of a sub-array
  - Swapping two elements of the array

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>27</td>
<td>29</td>
<td>23</td>
<td>19</td>
<td>13</td>
<td>31</td>
<td>15</td>
<td>17</td>
<td>21</td>
<td>25</td>
<td>11</td>
</tr>
</tbody>
</table>

The algorithm is on its fifth iteration
Find the smallest element in arr[4:15]
Swap smallest and first unsorted elements
void selectionSort(int arr[], int n){
    for(int i = 0; i < n-1; ++i){
        int smallest = getSmallest(arr, i, n);
        swap(arr, i, smallest);
    }
}

int getSmallest(int arr[], int start, int end){
    int smallest = start;
    for(int i = start + 1; i < end; ++i){
        if(arr[i] < arr[smallest]){
            smallest = i;
        }
    }
    return smallest;
}

void swap(int arr[], int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
void selectionSort(int arr[], int n){
    for(int i = 0; i < n-1; ++i){
        int smallest = getSmallest(arr, i, n);
        swap(arr, i, smallest);
    }
}

int getSmallest(int arr[], int start, int end){
    int smallest = start;
    for(int i = start + 1; i < end; ++i){
        if(arr[i] < arr[smallest]){
            smallest = i;
        }
    }
    return smallest;
}

void swap(int arr[], int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
void selectionSort(int arr[], int n){
    for(int i = 0; i < n-1; ++i){
        int smallest = getSmallest(arr, i, n);
        swap(arr, i, smallest);
    }
}

int getSmallest(int arr[], int start, int end){
    int smallest = start;
    for(int i = start + 1; i < end; ++i){
        if(arr[i] < arr[smallest]){
            smallest = i;
        }
    }
    return smallest;
}

4(start-end-1)+4 operations
we can substitute n for end
the start index is the index of i in the calling function
the values of which are the sequence {0,1,2,..., n-2}
1:n-1, 2:n-1, ..., n-1:n-1,
average = (n-1+1)/2 = n/2
average cost = 4(n/2)+4
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i) {
        int smallest = getSmallest(arr, i, n);
        swap(arr, i, smallest);
    }
}

for loop: 3(n-1)+2

Cost function: \( t_{\text{selection sort}} = 2n^2 - 2n + 10(n-1) + 2 \)
The barometer operation for selection sort is in the loop that finds the smallest item

- Since operations in that loop are executed the greatest number of times

The loop contains four operations

- Compare $i$ to $end$
- Compare $arr[i]$ to smallest
- Change smallest
- Increment $i$

```c
int getSmallest(arr[], start, end)

smallest = start

for(i = start + 1; i < end; ++i)
    if(arr[i] < arr[smallest])
        smallest = i

return smallest
```
## Barometer Operations

<table>
<thead>
<tr>
<th>Unsorted elements</th>
<th>Barometer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>$n-1$</td>
<td>$n-2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$n(n-1)/2$</td>
<td></td>
</tr>
</tbody>
</table>
Selection Sort Cases

- How is selection sort affected by the organization of the input?
  - The only work that varies based on the input organization is whether or not smallest is assigned the value of \( arr[i] \)

- What is the worst case organization?

- What is the best case organization?

- The difference between best case and worst case is quite small
  - \((n-1)(3(n/2)) + 10(n-1) + 2\) in the best case and
  - \((n-1)(4(n/2)) + 10(n-1) + 2\) in the worst case
Selection Sort Summary

- Ignoring leading constants, selection sort performs the following work
  - \( n(n - 1)/2 \) barometer operations, regardless of the original order of the input
  - \( n - 1 \) swaps
- The number of comparisons dominates the number of swaps
- The organization of the input only affects the leading constant of the barometer operations
The array is divided into sorted part and unsorted parts

The sorted part is expanded one element at a time

- By moving elements in the sorted part up one position until the correct position for the first unsorted element is found
  - Note that the first unsorted element is stored so that it is not lost when it is written over by this process

- The first unsorted element is then copied to the insertion point

The algorithm is on its fourth iteration

Find the correct position for arr[4]

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr</td>
<td>5</td>
<td>11</td>
<td>27</td>
<td>31</td>
<td>21</td>
<td>1</td>
<td>29</td>
<td>23</td>
<td>19</td>
<td>13</td>
<td>7</td>
<td>15</td>
<td>17</td>
<td>9</td>
<td>25</td>
<td>3</td>
</tr>
</tbody>
</table>

temp: 21
Insertion Sort

- The array is divided into sorted part and unsorted parts
- The sorted part is expanded one element at a time
  - By moving elements in the sorted part up one position until the correct position for the first unsorted element is found
    - Note that the first unsorted element’s value is stored so that it is not lost when it is written over by this process
  - The first unsorted element is then copied to the insertion point

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr</td>
<td>5</td>
<td>11</td>
<td>21</td>
<td>27</td>
<td>31</td>
<td>1</td>
<td>29</td>
<td>23</td>
<td>19</td>
<td>13</td>
<td>7</td>
<td>15</td>
<td>17</td>
<td>9</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>temp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>21</td>
</tr>
</tbody>
</table>

The algorithm is on its fourth iteration
Find the correct position for arr[4]
Move up elements in the sorted part until the position for 21 is found
How much work was performed by expanding the sorted part of the array by one element?
- The value 21 was stored in a variable, \( temp \)
- The values 27 and 31 were compared to 21
  - And moved up one position in the array
- The value 11 was compared to 21, but not moved
- The value of \( temp \) was written to \( arr[2] \)

How much work will be performed expanding the sorted part of the array to include the value 1?
How much work will be performed expanding the sorted part of the array to include the value 29?
void insertionSort(int arr[], int n){
    for(int i = 1; i < n; ++i){
        temp = arr[i];
        int pos = i;
        // Shuffle up all sorted items > arr[i]
        while(pos > 0 && arr[pos - 1] > temp){
            arr[pos] = arr[pos - 1];
            pos--;
        } //while
        // Insert the current item
        arr[pos] = temp;
    }
}
void insertionSort(int arr[], int n)
{
    for(int i = 1; i < n; ++i)
    {
        temp = arr[i];
        int pos = i;
        // Shuffle up all sorted items > arr[i]
        while(pos > 0 && arr[pos - 1] > temp){
            arr[pos] = arr[pos - 1];
            pos--;
        } //while
        // Insert the current item
        arr[pos] = temp;
    }
}

What is the worst case organization?

outer loop runs \( n-1 \) times: \( n \times \frac{n - 1}{2} \)

pos ranges from 1 to \( n-1 \); \( n/2 \) on average

worst case: \( pos - 1 \) times for each iteration

inner loop body how many times?

outer loop n-1 times
## Insertion Sort Worst Case Cost

<table>
<thead>
<tr>
<th>Sorted Elements</th>
<th>Worst-case Search</th>
<th>Worst-case Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(n-1)</td>
<td>(n-1)</td>
<td>(n-1)</td>
</tr>
<tr>
<td>(n(n-1)/2)</td>
<td>(n(n-1)/2)</td>
<td></td>
</tr>
</tbody>
</table>
Insertion Sort Worst Case

- In the worst case the array is in reverse order
- Every item has to be moved all the way to the front of the array
  - The outer loop runs \( n-1 \) times
    - In the first iteration, one comparison and move
    - In the last iteration, \( n-1 \) comparisons and moves
    - On average, \( n/2 \) comparisons and moves
  - For a total of \( n \times (n-1) / 2 \) comparisons and moves
The efficiency of insertion sort is affected by the state of the array to be sorted.

What is the best case?

- In the best case the array is already completely sorted!
- No movement of any array element is required
- Requires $n$ comparisons
What is the average case cost?
- Is it closer to the best case?
- Or the worst case?

If *random* data is sorted, insertion sort is usually closer to the worst case
- Around $n \times (n-1) / 4$ comparisons

And what do we mean by average input for a sorting algorithm in anyway?
Introduction to QuickSort
QuickSort Introduction

- Quicksort is a more efficient sorting algorithm than either selection or insertion sort
  - It sorts an array by repeatedly *partitioning* it
- Partitioning is the process of dividing an array into sections (partitions), based on some criteria
  - Big and small values
  - Negative and positive numbers
  - Names that begin with *a-m*, names that begin with *n-z*
  - Darker and lighter pixels
Partition this array into *small* and *big* values using a partitioning algorithm

31 12 07 23 93 02 11 18
Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm.

We will partition the array around the last value (18), we'll call this value the *pivot*.

Use two indices, one at each end of the array, call them *low* and *high*.

```
31 12 07 23 93 02 11 18
```
Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm.

We will partition the array around the last value (18), we'll call this value the *pivot*.

Use two indices, one at each end of the array, call them *low* and *high*.

arr[*low*] (31) is greater than the pivot and should be on the right, we need to swap it with something.
Partitioning an Array

Partition this array into small and big values using a partitioning algorithm.

We will partition the array around the last value (18), we'll call this value the pivot.

Use two indices, one at each end of the array, call them low and high.

- arr[low] (31) is greater than the pivot and should be on the right, we need to swap it with something.
- arr[high] (11) is less than the pivot so swap with arr[low].
Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm.

We will partition the array around the last value (18), we'll call this value the *pivot*.

Use two indices, one at each end of the array, call them *low* and *high*.
Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm.

We will partition the array around the last value (18), we'll call this value the *pivot*.

Use two indices, one at each end of the array, call them *low* and *high*.

Partition this array into small and big values using a partitioning algorithm.

increment *low* until it needs to be swapped with something.

then decrement *high* until it can be swapped with *low*.
Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm.

We will partition the array around the last value (18), we'll call this value the *pivot*.

Use two indices, one at each end of the array, call them *low* and *high*.

Increment *low* until it needs to be swapped with something.

Then decrement *high* until it can be swapped with *low*.

And then swap them.
Partition this array into *small* and *big* values using a partitioning algorithm.

We will partition the array around the last value (18), we'll call this value the *pivot*.

Use two indices, one at each end of the array, call them *low* and *high*.

Partitioning Algorithm

Repeat this process until *high* and *low* are the same.
Partition this array into *small* and *big* values using a partitioning algorithm.

We will partition the array around the last value (18), we'll call this value the *pivot*.

Use two indices, one at each end of the array, call them *low* and *high*.

Repeat this process until *high* and *low* are the same.

We'd like the pivot value to be in the centre of the array, so we will swap it with the first item greater than it.
Partitioning an Array

Partition this array into small and big values using a partitioning algorithm.

We will partition the array around the last value (18), we'll call this value the pivot.

Use two indices, one at each end of the array, call them low and high.
Partitioning Question

Use the same algorithm to partition this array into small and big values.
Partitioning Question

Or this one:

09 08 07 06 05 04 02 01

01 08 07 06 05 04 02 09

smalls

pivot

bigs
Quicksort

- The quicksort algorithm works by repeatedly partitioning an array.
- Each time a sub-array is partitioned there is:
  - A sequence of small values,
  - A sequence of big values, and
  - A pivot value which is in the correct position.
- Partition the small values, and the big values:
  - Repeat the process until each sub-array being partitioned consists of just one element.
Quicksort Algorithm

- The quicksort algorithm repeatedly partitions an array until it is sorted
  - Until all partitions consist of at most one element
- A simple iterative approach would halve each sub-array to get partitions
  - But partitions are not necessarily the same size
  - So the start and end indexes of each partition are not easily predictable
Uneven Partitions

47 70 36 97 03 67 29 11 48 09 53

36 09 29 48 03 11 47 53 97 61 70

36 09 03 11 29 47 48 53 61 70 97

08 01 11 29 36 47 48 53 61 70 97

09 03 11 29 36 47 48 53 61 70 97

03 09 11 29 36 47 48 53 61 70 97
One way to implement quicksort might be to record the index of each new partition. But this is difficult and requires a reasonable amount of space.

- The goal is to record the start and end index of each partition.
- This can be achieved by making them the parameters of a recursive function.
void quicksort(arr[], int low, int high){
    if (low < high){
        pivot = partition(arr, low, high);
        quicksort(arr, low, pivot - 1);
        quicksort(arr, pivot + 1, high);
    }
}
How long does Quicksort take to run?
- Let's consider the best and the worst case
- These differ because the partitioning algorithm may not always do a good job

Let's look at the best case first
- Each time a sub-array is partitioned the pivot is the exact midpoint of the slice (or as close as it can get)
  - So it is divided in half
- What is the running time?
Quicksort Best Case

First partition

8  1  2  7  3  6  4  5

04 01 02 03 5 06 08 07

_smalls_

pivot

_bigs_
Quicksort Best Case

Second partition
Quicksort Best Case

Third partition

02 01 03 04 05 06 07 08

pivot1 done done done

01 02 03 04 05 06 07 08

pivot1
Assume the best case – each partition splits its sub-array in half

# of recursive calls

1

2

3

log₂(n)

qs(arr, 0, n-1)

sub-array size

n

n/2

n/4

each level entails approximately \( n \) operations

there are approximately \( \log₂(n) \) levels

approximately \( \log₂(n) \times n \) operations in total

John Edgar
QuickSort Best Case

- Each sub-array is divided in half in each partition
  - Each time a series of sub-arrays are partitioned $n$ (approximately) comparisons are made
  - The process ends once all the sub-arrays left to be partitioned are of size 1
- How many times does $n$ have to be divided in half before the result is 1?
  - $\log_2(n)$ times
  - Quicksort performs $n \times \log_2 n$ operations in the best case
Quicksort Worst Case

First partition

09 08 07 06 05 04 02 01

01 08 07 06 05 04 02 09

smalls

pivot

bigs
Quicksort Worst Case

Second partition

01 08 07 06 05 04 02 09

01 08 07 06 05 04 02 09

smalls

bigs

pivot
Quicksort Worst Case

Third partition

pivot

bigs

01 08 07 06 05 04 02 09

01 02 07 06 05 04 08 09
Quicksort Worst Case

Fourth partition

01 02 07 06 05 04 08 09

smalls

pivot
Quicksort Worst Case

Fifth partition

pivot

gigs
Sixth partition

Quicksort Worst Case

01 02 04 06 05 07 08 09

sort.pivot

01 02 04 06 05 07 08 09

sort.pivot

sort.sort

sort.sort

sort.sort

sort.sort

sort.sort

sort.sort
Quicksort Worst Case

Seventh partition!

01 02 04 06 05 07 08 09

pivot
Assume the worst case – each partition step results in a single sub-array

- # of recursive calls
  - 1
  - 2
  - 3
  - n

- sub-array size
  - n
  - n-1
  - n-2
  - 1

- qs(arr, 0, n-1)
  - qs(...)
  - qs(...)
  - ...
  - qs(...)

- each level entails on average \( n/2 \) operations
- there are approximately \( n \) levels
- approximately \( n^2/2 \) operations in total
Quicksort Worst Case

- Every partition step ends with no values on one side of the pivot
  - The array has to be partitioned $n$ times, not $\log_2(n)$ times
  - So in the worst case Quicksort performs around $n^2$ operations
- The worst case usually occurs when the array is nearly sorted (in either direction)
QuickSort Average Case

- With a large array we would have to be very, very unlucky to get the worst case
  - Unless there was some reason for the array to already be partially sorted
- The average case is much more like the best case than the worst case
- There is an easy way to fix a partially sorted arrays to that it is ready for quicksort
  - Randomize the positions of the array elements!
Comparisons
Calculation of a detailed cost function can be onerous and dependent on

- Exactly how the algorithm was implemented
  - Implementing selection sort as a single function would have resulted in a different cost function
- The definition of a single discrete operation
  - How many operations is this: \( mid = (low + high) / 2 \)?
- We are often interested in how algorithms behave as the problem size increases
There can be many ways to solve a problem
  - Different algorithms that produce the same result
    - There are numerous sorting algorithms
  - Compare algorithms by their behaviour for large input sizes, i.e., as $n$ gets large
    - On today’s hardware, *most* algorithms perform quickly for small $n$
  - Interested in growth rate as a function of $n$
    - Sum an array: *linear* growth
    - Sort with selection sort: *quadratic* growth
Measuring the performance of an algorithm can be simplified by
  - Only considering the highest order term
    - i.e. only consider the number of times that the barometer instruction is executed
  - And ignoring leading constants
Consider the selection sort algorithm
  - \( t_{\text{selection sort}} = 2n^2 - 2n + 10(n-1) + 2 \)
  - Its cost function approximates to \( n^2 \)
What are the approximate number of barometer operations for the algorithms we looked at?

- Ignoring leading constants
  - Linear search: $n$
  - Binary search: $\log_2 n$
  - Selection sort: $n^2$
  - Insertion sort: $n^2$
  - Quicksort: $n(\log_2(n))$
What do we want to know when comparing two algorithms?
- Often, the most important thing is how quickly the time requirements increase with input size
  - e.g. If we double the input size how much longer does an algorithm take?
- Here are some graphs ...
Small $n$

Note that $n$ is very small ...

Arbitrary functions for the sake of illustration
Larger $n$

Operations vs $n$

- $\log(n)$
- $200\log(n)$
- $n$
- $100n$
- $n\log(n)$
- $5n\log(n)$
- $n^2$
- $0.1n^2$
A graph showing the operations for different functions of $n$, with $n$ ranging from 1 to 900,000. The functions include $\log(n)$, $200\log(n)$, $n$, $100n$, $n\log(n)$, $5n\log(n)$, $n^2$, and $0.1n^2$. The graph illustrates how the operations grow much larger as $n$ increases, emphasizing the significant difference in growth rates.
Logarithmic Scale

Note pairs of growth rates a constant distance apart
Exact counting of operations is often difficult (and tedious), even for simple algorithms
- And is often not much more useful than estimates due to the relative importance of other factors

\textit{O Notation} is a mathematical language for evaluating the running-time of algorithms
- O-notation evaluates the \textit{growth rate} of an algorithm
Example of a Cost Function

Cost Function: \( t_A(n) = n^2 + 20n + 100 \)

- Which term in the function is the most important?
- It depends on the size of \( n \)
  - \( n = 2, \ t_A(n) = 4 + 40 + 100 \)
    - The constant, 100, is the dominating term
  - \( n = 10, \ t_A(n) = 100 + 200 + 100 \)
    - \( 20n \) is the dominating term
  - \( n = 100, \ t_A(n) = 10,000 + 2,000 + 100 \)
    - \( n^2 \) is the dominating term
  - \( n = 1000, \ t_A(n) = 1,000,000 + 20,000 + 100 \)
    - \( n^2 \) is still the dominating term
The general idea is ...

- Big-O notation does not give a precise formulation of the cost function for a particular data size.
- It expresses the general behaviour of the algorithm as the data size $n$ grows very large so ignores:
  - lower order terms and
  - constants
- A Big-O cost function is a simple function of $n$:
  - $n$, $n^2$, $\log_2(n)$, etc.
Express the number of operations in an algorithm as a function of $n$, the problem size

Briefly

- Take the dominant term
- Remove the leading constant
- Put $O(\ldots)$ around it

For example, $f(N) = 2n^2 - 2n + 10(n-1) + 2$

- i.e. $O(n^2)$
Of course leading constants matter

- Consider two algorithms
  - $f_1(n) = 20n^2$
  - $f_2(n) = 2n^2$
- Algorithm 2 runs ten times faster

Let's consider machine speed

- If machine 1 is ten times faster than machine 2 it will run the same algorithm ten times faster

Big O notation ignores leading constants

- It is a hardware independent analysis
O Notation, More Formally

- Given a function $T(n)$
  - Say that $T(N) = O(f(n))$ if $T(n)$ is at most a constant times $f(n)$
    - Except perhaps for some small values of $n$

- Properties
  - Constant factors don’t matter
  - Low-order terms don’t matter

- Rules
  - For any $k$ and any function $g(n)$, $k*g(n) = O(f(n))$
    - e.g., $5n = O(n)$
Why Big O?

- An algorithm is said to be order $f(n)$
  - Denoted as $O(f(n))$
- The function $f(n)$ is the algorithm's growth rate function
  - If a problem of size $n$ requires time proportional to $n$ then the problem is $O(n)$
    - e.g. If the input size is doubled so is the running time
An algorithm is order $f(n)$ if there are positive constants $k$ and $m$ such that

- $t_A(n) \leq k \times f(n)$ for all $n \geq m$
  - i.e. find constants $k$ and $m$ such that the cost function is less than or equal to $k \times$ a simpler function for all $n$ greater than or equal to $m$

- If so we would say that $t_A(n)$ is $O(f(n))$
Finding a constant \( k \mid t_A(n) \leq k \ast f(n) \) shows that \( t \) is \( O(f(n)) \)
  - e.g. If the cost function was \( n^2 + 20n + 100 \) and we believed this was \( O(n) \)
    - We claim to be able to find a constant \( k \mid t_A(n) \leq k \ast f(n) \) for all values of \( n \)
      - Which is not possible

For some small values of \( n \) lower order terms may dominate
  - The constant \( m \) addresses this issue
The idea is that a cost function can be approximated by another, simpler, function

- The simpler function has 1 variable, the data size $n$
- This function is selected such that it represents an upper bound on the value of $t_A(n)$

Saying that the time efficiency of algorithm $A$ $t_A(n)$ is $O(f(n))$ means that

- $A$ cannot take more than $O(f(n))$ time to execute, and
- The cost function $t_A(n)$ grows at most as fast as $f(n)$
An algorithm’s cost function is $3n + 12$
- If we can find constants $m$ and $k$ such that:
  - $k \times n \geq 3n + 12$ for all $n \geq m$ then
  - The algorithm is $O(n)$
- Find values of $k$ and $m$ so that this is true
  - $k = 4$, and
  - $m = 12$ then
  - $4n \geq 3n + 12$ for all $n \geq 12$
Another Big O Example

- A cost function is $2n^2 - 2n + 10(n-1) + 2$
- If we can find constants $m$ and $k$ such that:
  - $k \times n^2 \geq 2n^2 - 2n + 10(n-1) + 2$ for all $n \geq m$ then
  - The algorithm is $O(n^2)$
- Find values of $k$ and $m$ so that this is true
  - $k = 3$, and
  - $m = 9$ then
  - $3n^2 > 2n^2 - 2n + 10(n-1) + 2$ for all $n \geq 9$
This is the graph of the selection sort cost function and $3n^2$

Demonstrating that selection sort is $O(n^2)$

For all $n \geq 9$, $3(n^2) \geq 2n^2 - 2n + 10(n-1) + 2$

i.e. $k = 3$ and $m = 9$

After this point $3n^2$ is always going to be larger than $2n^2 - 2n + 10(n-1) + 2$
O Notation Examples

- All these expressions are $O(n)$
  - $n$, $3n$
  - $61n + 5$
  - $22n - 5$

- All these expressions are $O(n^2)$
  - $n^2$
  - $9n^2$
  - $18n^2 + 4n - 53$

- All these expressions are $O(n \log n)$
  - $n(\log n)$
  - $5n(\log 99n)$
  - $18 + (4n - 2)(\log (5n + 3))$
Arithmetic and O Notation

- $O(k \times f) = O(f)$ if $k$ is a constant
  - e.g. $O(23 \times O(\log n))$, simplifies to $O(\log n)$
- $O(f + g) = \max[O(f), O(g)]$
  - $O(n + n^2)$, simplifies to $O(n^2)$
- $O(f \times g) = O(f) \times O(g)$
  - $O(m \times n)$, equals $O(m) \times O(n)$
  - Unless there is some known relationship between $m$ and $n$
    that allows us to simplify it, e.g. $m < n$
Growth rate functions are typically one of the following:

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \times \log n)$
- $O(n^2)$
- $O(n^k)$
- $O(2^n)$
We write $O(1)$ to indicate something that takes a constant amount of time

- Array look up
- Swapping two values in an array
- Finding the minimum element of an *ordered* array takes $O(1)$ time
  - The minimum value is either the first or the last element of the array
- Binary or linear search best case

**Important**: constants can be large

- So in practice $O(1)$ is not *necessarily* efficient
- It tells us is that the algorithm will run at the same speed no matter the size of the input we give it
$O(\log n)$ – Logarithmic Time

- $O(\log n)$ algorithms run in *logarithmic* time
  - Binary search average and worst case
  - The logarithm is assumed to be base 2 unless specified otherwise
- Doubling the size of $n$ doubles increases the number of operations by one
- Algorithms might be $O(\log n)$
  - If they are divide and conquer algorithms that halve the search space for each loop iteration or recursive call
O(n) – Linear Time

- O(n) algorithms run in *linear* time
  - Linear search
  - Summing the contents of an array
  - Traversing a linked list
  - Insertion sort or bogo sort best case
- Doubling the size of n doubles the number of operations
- Algorithms might be O(n)
  - If they contain a single loop
    - That iterates from 0 to n (or some variation)
  - Make sure that loops only contain constant time operations
    - And evaluate any function calls
$O(n\log n)$

- $O(n\log n)$
  - Mergesort in all cases
  - Heap sort in all cases
  - Quicksort in the average and best case
  - $O(n\log n)$ is the best case for comparison sorts
    - We will not prove this in CMPT 225
- Growth rate is faster than linear but still slow compared to $O(n^2)$
- Algorithms are $O(n\log n)$
  - If they have one process that is linear that is repeated $O(\log n)$ times
$O(n^2)$ – Quadratic Time

- $O(n^2)$ algorithms run in *quadratic* time
  - Selection sort in all cases
  - Insertion sort in the average and worst case
  - Bubble sort in all cases
  - Quicksort worst case
- Doubling the size of $n$ quadruples the number of operations
- Algorithms might be $O(n^2)$
  - If they contain nested loops
    - As usual make sure to check the number of iterations in such loops
    - And that the loops do not contain non-constant function calls
$O(n^k)$ – Polynomial Time

- $O(n^k)$ algorithms are referred to as running in *polynomial* time
  - If $k$ is large they can be very slow
- Doubling the size of $n$ increases the number of operations by $2^k$
\( O(2^n) \) or \( O(k^n) \) algorithms are referred to as running in exponential time

Very slow, and if there is no better algorithm implies that the problem is intractable

- That is, problems of any reasonable size cannot be solved in a reasonable amount of time
  - Such as over the lifetime of the human race ...

\( O(n!) \) algorithms are even slower

- Traveling Salesman Problems (and many others)
- Bogo sort in the average case
The $O$ notation growth rate of some algorithms varies depending on the input.

Typically we consider three cases:

- **Worst case**, usually (relatively) easy to calculate and therefore commonly used.
- **Average case**, often difficult to calculate.
- **Best case**, usually easy to calculate but less important than the other cases.
  - Relying on the input having the best case organization is not generally a good idea.
**Other Related Notations**

- **Ω (omega) notation**
  - Gives a lower bound
    - Whereas O notation is an upper bound
  - There exists some constant $k$, such that $k \cdot f(n)$ is a lower bound on the cost function $g(n)$

- **Θ (theta) notation**
  - Gives an upper and lower bound
    - $k_1 \cdot f(n)$ is an upper bound on $g(n)$ and $k_2 \cdot f(n)$ is a lower bound on $g(n)$
Upper and Lower Bound

Operations vs. \( n \):

- \( O(n^2) \)
- \( \Omega(n^2) \)
- \( 2n^2 - 2n + 10(n-1) + 2 \)
- \( 3n^2 \)
- \( 2n^2 \)