Computer Science I

Searching & Sorting

Dr. Chris Bourke
cbourke@cse.unl.edu

Outline

Part I: Introduction & Linear Search

- Processing data is a fundamental operation in Computer Science

Introduction

- Processing data is a fundamental operation in Computer Science
- Two fundamental operations in processing data are *searching* and *sorting*

Introduction

- Processing data is a fundamental operation in Computer Science
- Two fundamental operations in processing data are *searching* and *sorting*
- Form the basis or preprocessing step of many algorithms

- Processing data is a fundamental operation in Computer Science
- Two fundamental operations in processing data are *searching* and *sorting*
- Form the basis or preprocessing step of many algorithms
- Large variety of algorithms have been developed

Searching

- Given a *collection* of *elements* $A = \{a_1, a_2, \ldots, a_n\}$ and a *key* k, find an element that "matches" k

Searching

- Given a *collection* of *elements* $A = \{a_1, a_2, \ldots, a_n\}$ and a *key* k, find an element that "matches" k
- Collection: haystack, key: needle

Very general problems statement:

- Collection: arrays, sets, lists, etc.

Very general problems statement:

- Collection: arrays, sets, lists, etc.
- Elements: integers, strings, structures, etc.

Searching

Very general problems statement:

- Collection: arrays, sets, lists, etc.
- Elements: integers, strings, structures, etc.
- "matches": could be any criteria!

Searching

Very general problems statement:

- Collection: arrays, sets, lists, etc.
- Elements: integers, strings, structures, etc.
- "matches": could be any criteria!
- Variations:

Very general problems statement:

- Collection: arrays, sets, lists, etc.
- Elements: integers, strings, structures, etc.
- "matches": could be any criteria!
- Variations:
  - Find the first/last such element

Very general problems statement:

- Collection: arrays, sets, lists, etc.
- Elements: integers, strings, structures, etc.
- “matches”: could be any criteria!
- Variations:
  - Find the first/last such element
  - Find all such elements

Very general problems statement:

- Collection: arrays, sets, lists, etc.
- Elements: integers, strings, structures, etc.
- "matches": could be any criteria!
- Variations:
  - Find the first/last such element
  - Find all such elements
  - Find extremal elements

Searching

Very general problems statement:

- Collection: arrays, sets, lists, etc.
- Elements: integers, strings, structures, etc.
- "matches": could be any criteria!
- Variations:
  - Find the first/last such element
  - Find all such elements
  - Find extremal elements
- What do you do for unsuccessful searches?

Potential Solution: Linear Search

- Basic idea: iterate through each element

Linear Search

Potential Solution: Linear Search

- Basic idea: iterate through each element
- For each element, apply the "matching" criteria

Linear Search

Potential Solution: Linear Search

- Basic idea: iterate through each element
- For each element, apply the "matching" criteria
- Stop at the first match

Potential Solution: Linear Search

- Basic idea: iterate through each element
- For each element, apply the "matching" criteria
- Stop at the first match
- If no such element, return a "flag" value

A potential C solution:

- Take an array of integers

A potential C solution:

- Take an array of integers
- An integer key k

A potential C solution:

- Take an array of integers
- An integer key k
- Find the first element equal to k

A potential C solution:

- Take an array of integers
- An integer key k
- Find the first element equal to k
- Return its index

Linear Search

A potential C solution:

- Take an array of integers
- An integer key k
- Find the first element equal to k
- Return its index
- Unsuccessful search: -1 as a flag value

Linear Search


```c
/**
 * This function takes an array of integers
 * and searches it for the given key, returning
 * the index at which it finds it, or -1 if no
 * such element exists.
 */
int linearSearch(const int *arr, int n, int key) {

  for(int i=0; i<n; i++) {
    if(arr[i] == key) {
      //you found your needle...
      return i;
    }
  }
  //the needle was not found
  return -1;
}
```

Linear Search: Observations

Solution works

- Solution works but is less than ideal

Linear Search: Observations

Solution works

- Solution works but is less than ideal
- It only applies to arrays of integers

Solution works

- Solution works but is less than ideal
- It only applies to arrays of integers
- Search arrays of `double` or strings or `Student` structures, etc.: copy-pasta

Linear Search: Observations

Solution works

- Solution works but is less than ideal
- It only applies to arrays of integers
- Search arrays of `double` or strings or `Student` structures, etc.: copy-pasta
- Different search criteria (search `Student` by NUID or name): yet another implementation

Solution works

- Solution works but is less than ideal
- It only applies to arrays of integers
- Search arrays of `double` or strings or `Student` structures, etc.: copy-pasta
- Different search criteria (search `Student` by NUID or name): yet another implementation
- Ultimate goal: one single "generic" searching (and sorting) solution that will work with arrays of any type of data

Linear Search: Observations

Solution works

- Solution works but is less than ideal
- It only applies to arrays of integers
- Search arrays of `double` or strings or `Student` structures, etc.: copy-pasta
- Different search criteria (search `Student` by NUID or name): yet another implementation
- Ultimate goal: one single "generic" searching (and sorting) solution that will work with arrays of any type of data
- Can we do better?

Part II: Binary Search & Comparison

Binary Search: Basic Idea

- Can we do better than linear search?

Binary Search: Basic Idea

- Can we do better than linear search?
- Suppose that the array is *sorted*: how might we exploit that structure?

Binary Search: Basic Idea

- Can we do better than linear search?
- Suppose that the array is *sorted*: how might we exploit that structure?
- Searching for an element k

Binary Search: Basic Idea

- Can we do better than linear search?
- Suppose that the array is *sorted*: how might we exploit that structure?
- Searching for an element k
- Examine the middle element, m:

Binary Search: Basic Idea

- Can we do better than linear search?
- Suppose that the array is *sorted*: how might we exploit that structure?
- Searching for an element k
- Examine the middle element, m:
  - If $m = k$: success!

Binary Search: Basic Idea

- Can we do better than linear search?
- Suppose that the array is *sorted*: how might we exploit that structure?
- Searching for an element k
- Examine the middle element, m:
  - If $m = k$: success!
  - If $k < m$: k must lie in the left-half of the array

Binary Search: Basic Idea

- Can we do better than linear search?
- Suppose that the array is *sorted*: how might we exploit that structure?
- Searching for an element k
- Examine the middle element, m:
  - If $m = k$: success!
  - If $k < m$: k must lie in the left-half of the array
  - If $m < k$: k must lie in the right-half of the array

Illustration

$$\underbrace{\boxed{a_1}\boxed{\quad\cdots\quad}\boxed{a_{\frac{n}{2}-1}}}_{k<m}\underset{\substack{\uparrow\\ m}}{\boxed{a_{\frac{n}{2}}}}\underbrace{\boxed{a_{\frac{n}{2}+1}}\boxed{\quad\cdots\quad}\boxed{a_n}}_{m<k}$$

Example

Search for $k = 42$:

$$l = 0, \quad r = 10, \quad m = 5$$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

Example

Search for $k = 42$:

$$l = 0, \quad r = 10, \quad m = 5$$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

Example

$12 < 42 = k$:

$$l = 6, \quad r = 10, \quad m = 5$$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

Example

$$l = 6, \quad r = 10, \quad m = 8$$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

Example

$$l = 6, \quad r = 10, \quad m = 8$$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

Example

$42 < 102$:

$$l = 6, \quad r = 7, \quad m = 8$$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

Example

$$l = 6, \quad r = 7, \quad m = 6$$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

Example

$34 < 42 = k$:

$$l = 6, \quad r = 7, \quad m = 6$$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

Example

$$l = 7, \quad r = 7, \quad m = 7$$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

Example

$a_7 = 42 = k$

$$l = 7, \quad r = 7, \quad m = 7$$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

Recursive Code

```c
int binarySearch(const int *arr, int l, int r, int k) {
  if(l > r) {
    return -1;
  } else {
    int m = (l + r) / 2; //bad in practice

    if(arr[m] == k) {
      return m;
    } else if(k < arr[m]) {
      return binarySearch(arr, l, m-1, k);
    } else if(arr[m] < k) {
      return binarySearch(arr, m+1, r, k);
    }
  }
}
```

Iterative Code


```c
int binarySearch(const int *arr, int n, int k) {
  int l = 0;
  int r = n-1;
  while(l <= r) {
    int m = (l + r) / 2; //bad in practice

    if(arr[m] == k) {
      return m;
    } else if(k < arr[m]) {
      r = m - 1;
    } else if(arr[m] < k) {
      l = m+1;
    }
  }
  return -1;
}
```

Analysis

- Which is better? How much better?

Analysis

- Which is better? How much better?
- How much "work" does each algorithm perform?

- Which is better? How much better?
- How much “work” does each algorithm perform?
- Suppose we search an array of n elements

Analysis

- Which is better? How much better?
- How much "work" does each algorithm perform?
- Suppose we search an array of n elements
- How many *comparisons* does each search perform?

Linear Search Analysis

- Best case scenario: you get lucky and immediately find the element, making one single comparison

Linear Search Analysis

- Best case scenario: you get lucky and immediately find the element, making one single comparison
- Worst Case: you are unlucky and make all n comparisons

Linear Search Analysis

- Best case scenario: you get lucky and immediately find the element, making one single comparison
- Worst Case: you are unlucky and make all n comparisons
- Average case scenario: $\approx \frac{n}{2}$ comparisons

Linear Search Analysis

- Best case scenario: you get lucky and immediately find the element, making one single comparison
- Worst Case: you are unlucky and make all n comparisons
- Average case scenario: $\approx \frac{n}{2}$ comparisons
- Called *linear search* because the work is *linearly* proportional to the array size

Binary Search

- Worst case scenario: unsuccessful search

Binary Search

- Worst case scenario: unsuccessful search
- Or: when the list size is cut down to size 1

Binary Search

- Worst case scenario: unsuccessful search
- Or: when the list size is cut down to size 1
- Each comparison cuts the array (roughly) in half

Binary Search

- Worst case scenario: unsuccessful search
- Or: when the list size is cut down to size 1
- Each comparison cuts the array (roughly) in half
- After first iteration:

$$\frac{n}{2}$$

- Worst case scenario: unsuccessful search
- Or: when the list size is cut down to size 1
- Each comparison cuts the array (roughly) in half
- After first iteration:

$$\frac{n}{2}$$

- After second:

$$\frac{n}{4}$$

- After third:

$$\frac{n}{8}$$

Introduction
Binary Search
Analysis
Selection Sort
Quick Sort
Sorting in Practice
Function Pointers
Searching & Sorting in C

- After third:

$$\frac{n}{8}$$

- After k iterations:

$$\frac{n}{2^k}$$

- After third:

$$\frac{n}{8}$$

- After k iterations:

$$\frac{n}{2^k}$$

- Stops when

$$\frac{n}{2^k} = 1$$

Binary Search

- After third:
$$\frac{n}{8}$$
- After k iterations:
$$\frac{n}{2^k}$$
- Stops when
$$\frac{n}{2^k} = 1$$
- Solve for k:
$$k = \log_2(n)$$

Binary Search

- After third:
$$\frac{n}{8}$$
- After k iterations:
$$\frac{n}{2^k}$$
- Stops when
$$\frac{n}{2^k} = 1$$
- Solve for k:
$$k = \log_2(n)$$
- Roughly only $\log_2(n)$ comparisons are made.

Comparison

- Linear: $\approx n$ versus Binary Search: $\log_2(n)$

Comparison

- Linear: $\approx n$ versus Binary Search: $\log_2(n)$
- Linear search is *exponentially worse*

Comparison

- Linear: $\approx n$ versus Binary Search: $\log_2{(n)}$
- Linear search is *exponentially worse*
- Binary search is *exponentially faster*

Perspective

- Suppose we have a database of 1 trillion, 10^{12} elements

Perspective

- Suppose we have a database of 1 trillion, 10^{12} elements
- Unsorted using linear search:

$$\approx 5 \times 10^{11}$$

  comparisons

Perspective

- Suppose we have a database of 1 trillion, 10^{12} elements
- Unsorted using linear search:
$$\approx 5 \times 10^{11}$$
comparisons
- Sorted ("indexed") using binary search:
$$\approx \log_2 (10^{12}) \approx 40$$
comparisons

- Suppose we *double* the input size: $n \to 2n$

Another Perspective
Growth Rate

- Suppose we *double* the input size: $n \to 2n$
- Linear search would require $n \to 2n$ comparisons

Another Perspective

Growth Rate

- Suppose we *double* the input size: $n \rightarrow 2n$
- Linear search would require $n \rightarrow 2n$ comparisons
- Doubling the input size doubles the number of comparisons

- Suppose we *double* the input size: $n \to 2n$
- Linear search would require $n \to 2n$ comparisons
- Doubling the input size doubles the number of comparisons
- Binary search:

$$\log_2(n) \to \log_2(2n)$$

Another Perspective

Growth Rate

- Suppose we *double* the input size: $n \to 2n$
- Linear search would require $n \to 2n$ comparisons
- Doubling the input size doubles the number of comparisons
- Binary search:

$$\log_2(n) \to \log_2(2n)$$

- $\log_2(2n) = \log_2(n) + 1$

Another Perspective

Growth Rate

- Suppose we *double* the input size: $n \to 2n$
- Linear search would require $n \to 2n$ comparisons
- Doubling the input size doubles the number of comparisons
- Binary search:
$$\log_2(n) \to \log_2(2n)$$
- $\log_2(2n) = \log_2(n) + 1$
- Doubling the input size only adds one more comparison!

Part III: Selection Sort

Introduction

- To exploit binary search we need to be able to sort

Introduction

- To exploit binary search we need to be able to sort
- Many different sorting algorithms each with different properites

Introduction

- To exploit binary search we need to be able to sort
- Many different sorting algorithms each with different properites
- Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Tim Sort, etc.

- To exploit binary search we need to be able to sort
- Many different sorting algorithms each with different properites
- Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Tim Sort, etc.
- Some efficient, some inefficient

- To exploit binary search we need to be able to sort
- Many different sorting algorithms each with different properites
- Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Tim Sort, etc.
- Some efficient, some inefficient
- Start with a simple implementation: Selection Sort

Basic Idea

- Search through the array and find the minimal element

Basic Idea

- Search through the array and find the minimal element
- Swap it with the first element

Basic Idea

- Search through the array and find the minimal element
- Swap it with the first element
- Proceed with the remainder of the array

Basic Idea

- Search through the array and find the minimal element
- Swap it with the first element
- Proceed with the remainder of the array
- In general:

Basic Idea

- Search through the array and find the minimal element
- Swap it with the first element
- Proceed with the remainder of the array
- In general:
  - i-th iteration: find minimal element in `arr[i]` through `arr[n-1]`

Basic Idea

- Search through the array and find the minimal element
- Swap it with the first element
- Proceed with the remainder of the array
- In general:
  - i-th iteration: find minimal element in `arr[i]` through `arr[n-1]`
  - Swap it with `arr[i]`

Basic Idea

- Search through the array and find the minimal element
- Swap it with the first element
- Proceed with the remainder of the array
- In general:
  - i-th iteration: find minimal element in `arr[i]` through `arr[n-1]`
  - Swap it with `arr[i]`
  - Stop at $i = n - 1$ (last element is already sorted)

Basic Idea

- Search through the array and find the minimal element
- Swap it with the first element
- Proceed with the remainder of the array
- In general:
  - i-th iteration: find minimal element in `arr[i]` through `arr[n-1]`
  - Swap it with `arr[i]`
  - Stop at $i = n - 1$ (last element is already sorted)
- Demonstration

Selection Sort Example

Iteration 1

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

Selection Sort Example

Iteration 1

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

Selection Sort Example
Iteration 1

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

Selection Sort Example

Iteration 1

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

Selection Sort Example

Iteration 1

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

Selection Sort Example

Iteration 1

swap

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

Selection Sort Example

Iteration 1

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 1

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

Introduction
Binary Search
Selection Sort
Quick Sort
Sorting in Practice
Function Pointers
Searching & Sorting in C

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Introduction
Binary Search
Selection Sort
Quick Sort
Sorting in Practice
Function Pointers
Searching & Sorting in C

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

swap

index	0	1	2	3	4	5	6	7
contents	2	42	62	7	20	102	34	47

Selection Sort Example

Iteration 2

index	0	1	2	3	4	5	6	7
contents	2	7	62	42	20	102	34	47

Selection Sort Example

Iteration 2

index	0	1	2	3	4	5	6	7
contents	2	7	62	42	20	102	34	47

```c
void selectionSort(int *arr, int n) {

  for(int i=0; i<n-1; i++) {
    int minIndex = i;
    for(int j=i+1; j<n; j++) {
      if(arr[j] < arr[minIndex]) {
        minIndex = j;
      }
    }
    //swap
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

Analysis

- Selection sort is simple, but naive and inefficient

Analysis

- Selection sort is simple, but naive and inefficient
- How bad is it?

Analysis

- Selection sort is simple, but naive and inefficient
- How bad is it?
- How many comparisons does selection sort make on an array of size n?

- Selection sort is simple, but naive and inefficient
- How bad is it?
- How many comparisons does selection sort make on an array of size n?
  - First iteration: $n - 1$ comparisons

Analysis

- Selection sort is simple, but naive and inefficient
- How bad is it?
- How many comparisons does selection sort make on an array of size n?
  - First iteration: $n - 1$ comparisons
  - Second iteration: $n - 2$ comparisons

Analysis

- Selection sort is simple, but naive and inefficient
- How bad is it?
- How many comparisons does selection sort make on an array of size n?
  - First iteration: $n-1$ comparisons
  - Second iteration: $n-2$ comparisons
  - i-th iteration: $n-i$ comparisons

Analysis

- Selection sort is simple, but naive and inefficient
- How bad is it?
- How many comparisons does selection sort make on an array of size n?
  - First iteration: $n-1$ comparisons
  - Second iteration: $n-2$ comparisons
  - i-th iteration: $n-i$ comparisons
  - Last iteration: 1 comparison

Analysis

- Selection sort is simple, but naive and inefficient
- How bad is it?
- How many comparisons does selection sort make on an array of size n?
  - First iteration: $n-1$ comparisons
  - Second iteration: $n-2$ comparisons
  - i-th iteration: $n-i$ comparisons
  - Last iteration: 1 comparison
  - In total:

$$1+2+3+\cdots+(n-2)+(n-1)=\frac{n(n-1)}{2}=\frac{1}{2}n^2+\frac{1}{2}n$$

Perspective

- Selection sort is a *quadratic*, $\approx n^2$ sorting algorithm

Perspective

- Selection sort is a *quadratic*, $\approx n^2$ sorting algorithm
- How bad is this?

Perspective

- Selection sort is a *quadratic*, $\approx n^2$ sorting algorithm
- How bad is this?
- Sorting the database of 1 trillion, 10^{12} elements requires

$$\approx 5 \times 10^{23}$$

Perspective

- Selection sort is a *quadratic*, $\approx n^2$ sorting algorithm
- How bad is this?
- Sorting the database of 1 trillion, 10^{12} elements requires

$$\approx 5 \times 10^{23}$$

- 500 "Sextillion" comparisons

Perspective

- Selection sort is a *quadratic*, $\approx n^2$ sorting algorithm
- How bad is this?
- Sorting the database of 1 trillion, 10^{12} elements requires

$$\approx 5 \times 10^{23}$$

- 500 "Sextillion" comparisons
- NVIDIA GTX 1080Ti: 11.3 TeraFLOPS

Perspective

- Selection sort is a *quadratic*, $\approx n^2$ sorting algorithm
- How bad is this?
- Sorting the database of 1 trillion, 10^{12} elements requires

$$\approx 5 \times 10^{23}$$

- 500 "Sextillion" comparisons
- NVIDIA GTX 1080Ti: 11.3 TeraFLOPS

$$\frac{5 * 10^{23} \text{ operations}}{11.3 * 10^{12} \text{ ops/sec}} = 1,402.157 \textbf{ years}$$

Perspective

- Selection sort is a *quadratic*, $\approx n^2$ sorting algorithm
- How bad is this?
- Sorting the database of 1 trillion, 10^{12} elements requires

$$\approx 5 \times 10^{23}$$

- 500 "Sextillion" comparisons
- NVIDIA GTX 1080Ti: 11.3 TeraFLOPS

$$\frac{5 * 10^{23} \text{ operations}}{11.3 * 10^{12} \text{ ops/sec}} = 1,402.157 \textbf{ years}$$

- Not feasible for even "moderately large" inputs

Another Perspective

- Double the size of the array: $n \to 2n$

Another Perspective

- Double the size of the array: $n \to 2n$
- Number of comparisons grows:

$$n^2 \to (2n)^2 = 4n^2$$

Another Perspective

- Double the size of the array: $n \rightarrow 2n$
- Number of comparisons grows:

$$n^2 \rightarrow (2n)^2 = 4n^2$$

- Doubling the input *quadruples* the number of operations

Another Perspective

- Double the size of the array: $n \to 2n$
- Number of comparisons grows:

$$n^2 \to (2n)^2 = 4n^2$$

- Doubling the input *quadruples* the number of operations
- Four times slower!

Part IV: Quick Sort

- We need a better, more efficient sorting algorithm

- We need a better, more efficient sorting algorithm
- Lots exist, focus on Quick Sort

- We need a better, more efficient sorting algorithm
- Lots exist, focus on Quick Sort
- High level description only

- We need a better, more efficient sorting algorithm
- Lots exist, focus on Quick Sort
- High level description only
- Many variations of the same idea

- We need a better, more efficient sorting algorithm
- Lots exist, focus on Quick Sort
- High level description only
- Many variations of the same idea
- Basic Divide & Conquer strategy

Basic Idea

- Choose a *pivot* element

Basic Idea

- Choose a *pivot* element
- Partition elements around this pivot

Basic Idea

- Choose a *pivot* element
- Partition elements around this pivot
- Smaller elements to the left

Basic Idea

- Choose a *pivot* element
- Partition elements around this pivot
- Smaller elements to the left
- Larger elements to the right

Basic Idea

- Choose a *pivot* element
- Partition elements around this pivot
- Smaller elements to the left
- Larger elements to the right
- Place the pivot in the middle

Basic Idea

- Choose a *pivot* element
- Partition elements around this pivot
- Smaller elements to the left
- Larger elements to the right
- Place the pivot in the middle
- Pivot ends up where it should be

Basic Idea

- Choose a *pivot* element
- Partition elements around this pivot
- Smaller elements to the left
- Larger elements to the right
- Place the pivot in the middle
- Pivot ends up where it should be
- Recursively run quick sort on the left and right halves

Basic Idea

- Choose a *pivot* element
- Partition elements around this pivot
- Smaller elements to the left
- Larger elements to the right
- Place the pivot in the middle
- Pivot ends up where it should be
- Recursively run quick sort on the left and right halves
- Demonstration

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

$pivot = 42$								

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

$pivot = 42$								

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

$pivot = 42$								
	2							

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47
$pivot = 42$	2							62

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

$pivot = 42$	2	7						62

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47
$pivot = 42$	2	7	20					62

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47
$pivot = 42$	2	7	20				102	62

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47
$pivot = 42$	2	7	20	34			102	62

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

$pivot = 42$	2	7	20	34		47	102	62

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

$pivot = 42$	2	7	20	34	42	47	102	62

Quick Sort Example

index	0	1	2	3	4	5	6	7
contents	42	2	62	7	20	102	34	47

$pivot = 42$	2	7	20	34	42	47	102	62

```
quickSort(arr, 0, 3);
```

```
quickSort(arr, 5, 7);
```

Analysis

- Best/Worst/Average case analysis

Analysis

- Best/Worst/Average case analysis
- Quick Sort makes roughly $n \log_2(n)$ comparisons

Analysis

- Best/Worst/Average case analysis
- Quick Sort makes roughly $n \log_2(n)$ comparisons
- *Much* better than n^2

Analysis

- Best/Worst/Average case analysis
- Quick Sort makes roughly $n \log_2(n)$ comparisons
- *Much* better than n^2
- Comparisons

Analysis

- Sorting the database of 1 trillion, 10^{12} records

Analysis

- Sorting the database of 1 trillion, 10^{12} records
- Comparisons:

$$10^{12} \cdot \log_2 10^{12} \approx 4 \times 10^{13}$$

Analysis

- Sorting the database of 1 trillion, 10^{12} records
- Comparisons:

$$10^{12} \cdot \log_2 10^{12} \approx 4 \times 10^{13}$$

- 40 trillion comparisons

Analysis

- Sorting the database of 1 trillion, 10^{12} records
- Comparisons:

$$10^{12} \cdot \log_2 10^{12} \approx 4 \times 10^{13}$$

- 40 trillion comparisons
- NVIDIA GTX 1080Ti: 11.3 TeraFLOPS

Analysis

- Sorting the database of 1 trillion, 10^{12} records
- Comparisons:

$$10^{12} \cdot \log_2 10^{12} \approx 4 \times 10^{13}$$

- 40 trillion comparisons
- NVIDIA GTX 1080Ti: 11.3 TeraFLOPS

$$\frac{4 \times 10^{13} \text{ operations}}{11.3 * 10^{12} \text{ ops/sec}} = 3.5 \textbf{ seconds}$$

Analysis

- Sorting the database of 1 trillion, 10^{12} records
- Comparisons:

$$10^{12} \cdot \log_2 10^{12} \approx 4 \times 10^{13}$$

- 40 trillion comparisons
- NVIDIA GTX 1080Ti: 11.3 TeraFLOPS

$$\frac{4 \times 10^{13} \text{ operations}}{11.3 * 10^{12} \text{ ops/sec}} = 3.5 \textbf{ seconds}$$

- Very feasible

Analysis

- Consider doubling the input size: $n \rightarrow 2n$

Analysis

- Consider doubling the input size: $n \rightarrow 2n$
- Number of comparisons:

$$n \log_2{(n)} \rightarrow 2n \log_2{(2n)}$$

Analysis

- Consider doubling the input size: $n \rightarrow 2n$
- Number of comparisons:

$$n \log_2(n) \rightarrow 2n \log_2(2n)$$

- $2n \log_2(2n) = 2n \log_2(n) + 2n$

Analysis

- Consider doubling the input size: $n \to 2n$
- Number of comparisons:

$$n \log_2(n) \to 2n \log_2(2n)$$

- $2n \log_2(2n) = 2n \log_2(n) + 2n$
- Roughly only twice as many

Analysis

- Consider doubling the input size: $n \rightarrow 2n$
- Number of comparisons:

$$n \log_2(n) \rightarrow 2n \log_2(2n)$$

- $2n \log_2(2n) = 2n \log_2(n) + 2n$
- Roughly only twice as many
- Often referred to as *quasilinear*

Part V: Sorting in Practice

In Practice

- Don't "roll your own" searching/sorting algorithms

In Practice

- Don't "roll your own" searching/sorting algorithms
- Use standard library functions

In Practice

- Don't "roll your own" searching/sorting algorithms
- Use standard library functions
- *But*: we don't want dozens of different functions one for each type of variable or criteria that we want to sort with respect to

In Practice

- Don't "roll your own" searching/sorting algorithms
- Use standard library functions
- *But*: we don't want dozens of different functions one for each type of variable or criteria that we want to sort with respect to
- Want ONE generic solution that can sort *any* type of data by *any* criteria

In Practice

- Don't "roll your own" searching/sorting algorithms
- Use standard library functions
- *But*: we don't want dozens of different functions one for each type of variable or criteria that we want to sort with respect to
- Want ONE generic solution that can sort *any* type of data by *any* criteria
- One sorting function to sort them all

Comparators

- Solution: use one *generic* sorting function

Comparators

- Solution: use one *generic* sorting function
- Needs to know how to order two elements, a, b

Comparators

- Solution: use one *generic* sorting function
- Needs to know how to order two elements, a, b
- Are they in order or do they need to be swapped?

- Solution: use one *generic* sorting function
- Needs to know how to order two elements, a, b
- Are they in order or do they need to be swapped?
- Solution: A *comparator* function

- Solution: use one *generic* sorting function
- Needs to know how to order two elements, a, b
- Are they in order or do they need to be swapped?
- Solution: A *comparator* function
- Given two elements a, b it returns:

Comparators

- Solution: use one *generic* sorting function
- Needs to know how to order two elements, a, b
- Are they in order or do they need to be swapped?
- Solution: A *comparator* function
- Given two elements a, b it returns:
    - *something* negative if $a < b$

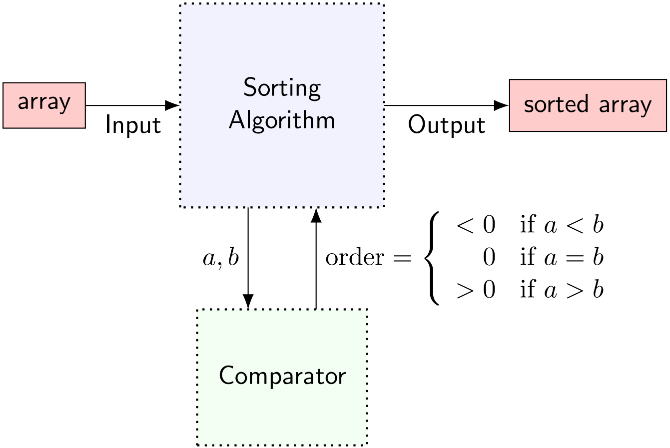
Comparators

- Solution: use one *generic* sorting function
- Needs to know how to order two elements, a, b
- Are they in order or do they need to be swapped?
- Solution: A *comparator* function
- Given two elements a, b it returns:
  - *something* negative if $a < b$
  - zero if $a = b$

Comparators

- Solution: use one *generic* sorting function
- Needs to know how to order two elements, a, b
- Are they in order or do they need to be swapped?
- Solution: A *comparator* function
- Given two elements a, b it returns:
  - *something* negative if $a < b$
  - zero if $a = b$
  - *something* positive if $a > b$

Comparator Illustration



Comparators in C

- In C, a *comparator function* has the following signature

Comparators in C

- In C, a *comparator function* has the following signature
- `int cmp(const void *a, const void *b);`

Comparators in C

- In C, a *comparator function* has the following signature
- `int cmp(const void *a, const void *b);`
- `const` means we won't change it, only compare it

Comparators in C

- In C, a *comparator function* has the following signature
- `int cmp(const void *a, const void *b);`
- `const` means we won't change it, only compare it
- `void *` is a *generic pointer* that can point to anything

Comparators in C

- In C, a *comparator function* has the following signature
- `int cmp(const void *a, const void *b);`
- `const` means we won't change it, only compare it
- `void *` is a *generic pointer* that can point to anything
- Recall: `malloc()`

Standard Pattern

Standard Pattern:

- Cast the `void *` to a particular data type

Standard Pattern:

- Cast the `void *` to a particular data type
- Use the data's *state* to determine the proper order

Standard Pattern

Standard Pattern:

- Cast the `void *` to a particular data type
- Use the data's *state* to determine the proper order
- Return an integer value that expresses the proper order

Best Practice

Best Practices:

- Use descriptive function names

Best Practice

Best Practices:

- Use descriptive function names
- Be explicit in your comparisons

Best Practice

Best Practices:

- Use descriptive function names
- Be explicit in your comparisons
- Avoid “tricks”

Best Practice

Best Practices:

- Use descriptive function names
- Be explicit in your comparisons
- Avoid "tricks"
- Reuse comparator functionality when possible

Examples

- Write a comparator to order integers in non-decreasing order
- Write a comparator to order integers in non-increasing order
- Write a comparator to order `Student` structures by NUID
- Write a comparator to order `Student` structures by GPA
- Write a comparator to order `Student` structures by last name/first name

Part VI: Function Pointers

Function Pointers

- Now that we have comparator functions: how do we pass them to a generic sorting function?

- Now that we have comparator functions: how do we pass them to a generic sorting function?
- Easy to pass variables by value or by reference

- Now that we have comparator functions: how do we pass them to a generic sorting function?
- Easy to pass variables by value or by reference
- How do we pass a function?

Function Pointers

- Now that we have comparator functions: how do we pass them to a generic sorting function?
- Easy to pass variables by value or by reference
- How do we pass a function?
- We need *function pointers*

- Recall: a *pointer* refers to a memory location

Function Pointers

- Recall: a *pointer* refers to a memory location
- What is stored in memory?

- Recall: a *pointer* refers to a memory location
- What is stored in memory?
- Variables, arrays, data, *everything*

- Recall: a *pointer* refers to a memory location
- What is stored in memory?
- Variables, arrays, data, *everything*
- A program's code is stored in memory, including its *functions*

Function Pointers

- Recall: a *pointer* refers to a memory location
- What is stored in memory?
- Variables, arrays, data, *everything*
- A program's code is stored in memory, including its *functions*
- We can create pointers that point to memory locations that contain functions!

Function Pointers

- Recall: a *pointer* refers to a memory location
- What is stored in memory?
- Variables, arrays, data, *everything*
- A program's code is stored in memory, including its *functions*
- We can create pointers that point to memory locations that contain functions!
- Function pointers allow us to "pass" a function to another function

Function Pointers

- Recall: a *pointer* refers to a memory location
- What is stored in memory?
- Variables, arrays, data, *everything*
- A program's code is stored in memory, including its *functions*
- We can create pointers that point to memory locations that contain functions!
- Function pointers allow us to "pass" a function to another function
- Called "callback" functions

Function Pointers

- Recall: a *pointer* refers to a memory location
- What is stored in memory?
- Variables, arrays, data, *everything*
- A program's code is stored in memory, including its *functions*
- We can create pointers that point to memory locations that contain functions!
- Function pointers allow us to "pass" a function to another function
- Called "callback" functions
- Demonstration

Function Pointers: Demo


```c
//create a pointer called ptrToFunc that can point to a
//function that returns an integer and takes three arguments:
//(int, double, char)

int (*ptrToFunc)(int, double, char) = NULL;

//declare a pointer that can point to math's sqrt function
double (*ptrToSqrt)(double) = NULL;

//let's make ptrToSqrt point to the sqrt function
ptrToSqrt = sqrt;

//you can call a function via its pointer:
double x = ptrToSqrt(2.0);

//careful: you can reassign standard library functions:
sqrt = sin;
double y = sqrt(3.14159); //0
//don't do this

//a function that takes another function:
void runAFunction(double x, double (*func)(double)) {
  //run func on x:
  double y = func(x);
}
```

Part VII: Searching & Sorting in C

- To make generic searching & sorting functions, we need to pass in a comparator

Searching & Sorting in C

- To make generic searching & sorting functions, we need to pass in a comparator
- Function pointers allow us to do this

- To make generic searching & sorting functions, we need to pass in a comparator
- Function pointers allow us to do this
- The array to be searched/sorted is also generic, `void *`

Searching & Sorting in C

- To make generic searching & sorting functions, we need to pass in a comparator
- Function pointers allow us to do this
- The array to be searched/sorted is also generic, `void *`
- Demonstration: generic linear search

Generic Linear Search


```c
/**
 * This function takes an array of integers
 * and searches it for the given key, returning
 * the index at which it finds it, or -1 if no
 * such element exists.
 */
int linearSearch(const void *key, const void *arr, int n, int size, int (*compar) (const

  for(int i=0; i<n; i++) {
    if(compar(key, (arr + i * size) ) == 0) {
      return i;
    }
  }
  return -1;
}
```

Sorting in C

- The standard C library provides a generic sorting function

```c
void qsort(void *base,
           size_t nel,
           size_t size,
           int (*compar)(const void *, const void *));
```

Sorting in C

- The standard C library provides a generic sorting function

```
void qsort(void *base,
           size_t nel,
           size_t size,
           int (*compar)(const void *, const void *));
```

- `base` is the array of elements to be sorted

Sorting in C

- The standard C library provides a generic sorting function

```
void qsort(void *base,
           size_t nel,
           size_t size,
           int (*compar)(const void *, const void *));
```

- `base` is the array of elements to be sorted
- `nel` is the number of elements in the array

Sorting in C

- The standard C library provides a generic sorting function

```
void qsort(void *base,
           size_t nel,
           size_t size,
           int (*compar)(const void *, const void *));
```

- `base` is the array of elements to be sorted
- `nel` is the number of elements in the array
- `size` is the number of bytes each element takes

Sorting in C

- The standard C library provides a generic sorting function

```
void qsort(void *base,
           size_t nel,
           size_t size,
           int (*compar)(const void *, const void *));
```

- `base` is the array of elements to be sorted
- `nel` is the number of elements in the array
- `size` is the number of bytes each element takes
- `compar` the comparator function you want to use to order elements

Sorting in C

- The standard C library provides a generic sorting function

```
void qsort(void *base,
           size_t nel,
           size_t size,
           int (*compar)(const void *, const void *));
```

- `base` is the array of elements to be sorted
- `nel` is the number of elements in the array
- `size` is the number of bytes each element takes
- `compar` the comparator function you want to use to order elements
- Demonstration

Binary Search in C

- The standard C library provides a generic binary search function

```
void * bsearch(const void *key,
               const void *base,
               size_t nel,
               size_t size,
               int (*compar) (const void *, const void *));
```

Binary Search in C

- The standard C library provides a generic binary search function

```
void * bsearch(const void *key,
               const void *base,
               size_t nel,
               size_t size,
               int (*compar) (const void *, const void *));
```

- Returns a pointer to the element that "matches" the key

Binary Search in C

- The standard C library provides a generic binary search function

```c
void * bsearch(const void *key,
               const void *base,
               size_t nel,
               size_t size,
               int (*compar) (const void *, const void *));
```

- Returns a pointer to the element that “matches” the key
- (an element such that the comparator returns 0)

- The standard C library provides a generic binary search function

```
void * bsearch(const void *key,
               const void *base,
               size_t nel,
               size_t size,
               int (*compar) (const void *, const void *));
```

- Returns a pointer to the element that “matches” the key
- (an element such that the comparator returns 0)
- Returns `NULL` if no such element

Binary Search in C

- The standard C library provides a generic binary search function

```c
void * bsearch(const void *key,
               const void *base,
               size_t nel,
               size_t size,
               int (*compar) (const void *, const void *));
```

- Returns a pointer to the element that "matches" the key
- (an element such that the comparator returns 0)
- Returns `NULL` if no such element
- Assumes the array is sorted in the same order as defined by `compar`

Binary Search in C

- The standard C library provides a generic binary search function

```c
void * bsearch(const void *key,
               const void *base,
               size_t nel,
               size_t size,
               int (*compar) (const void *, const void *));
```

- Returns a pointer to the element that "matches" the key
- (an element such that the comparator returns 0)
- Returns `NULL` if no such element
- Assumes the array is sorted in the same order as defined by `compar`
- Demonstration