CSE 373

MAY 26TH – NON-COMPARISON SORTING
ASSORTED MINUTIAE

• HW6 Out – Due next Wednesday
  • No Java Libraries
ASSORTED MINUTIAE

• HW6 Out – Due next Wednesday
  • No Java Libraries
• Two exam review sessions
  • Wednesday: 1:00 – 2:20 – CMU 120
  • Friday: 4:30 – 6:20 – EEB 105
TODAY

• Non-comparison sorts
SORTING

• “Slow” sorts
SORTING

• “Slow” sorts
  • Insertion
  • Selection
SORTING

• “Slow” sorts
  • Insertion
  • Selection

• “Fast” sorts
SORTING

• “Slow” sorts
  • Insertion
  • Selection
• “Fast” sorts
  • Quick
  • Merge
  • Heap
SORTING

• “Slow” sorts
  • Insertion
  • Selection

• “Fast” sorts
  • Quick
  • Merge
  • Heap

• These are all comparison sorts, can’t do better than $O(n \log n)$
SORTING

• Non-comparison sorts
SORTING

• Non-comparison sorts
  • If we know something about the data, we don't strictly need to compare objects to each other
SORTING

• Non-comparison sorts
  • If we know something about the data, we don’t strictly need to compare objects to each other
  • If there are only a few possible values and we know what they are, we can just sort by identifying the value
SORTING

• Non-comparison sorts
  • If we know something about the data, we don’t strictly need to compare objects to each other
  • If there are only a few possible values and we know what they are, we can just sort by identifying the value
  • If the data are strings and ints of finite length, then we can take advantage of their sorted order.
SORTING

- Two sorting techniques we use to this end
SORTING

- Two sorting techniques we use to this end
  - Bucket sort
SORTING

- Two sorting techniques we use to this end
  - Bucket sort
  - Radix sort
SORTING

• Two sorting techniques we use to this end
  • Bucket sort
  • Radix sort

• If the data is sufficiently structured, we can get $O(n)$ runtimes
BUCKETSORT

If all values to be sorted are known to be integers between 1 and $K$ (or any small range):

- Create an array of size $K$
- Put each element in its proper bucket (a.k.a. bin)
- If data is only integers, no need to store more than a count of how times that bucket has been used

Output result via linear pass through array of buckets

<table>
<thead>
<tr>
<th>count array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

- Example:
  - $K=5$
  - input (5,1,3,4,3,2,1,1,5,4,5)
  - output: 1,1,1,2,3,3,4,4,5,5,5
ANALYZING BUCKET SORT

Overall: $O(n+K)$
  - Linear in $n$, but also linear in $K$

Good when $K$ is smaller (or not much larger) than $n$
  - We don’t spend time doing comparisons of duplicates

Bad when $K$ is much larger than $n$
  - Wasted space; wasted time during linear $O(K)$ pass

For data in addition to integer keys, use list at each bucket
BUCKET SORT

Most real lists aren’t just keys; we have data
Each bucket is a list (say, linked list)
To add to a bucket, insert in $O(1)$ (at beginning, or keep pointer to last element)

Example: Movie ratings; scale 1-5

Input:
- 5: Casablanca
- 3: Harry Potter movies
- 5: Star Wars Original Trilogy
- 1: Rocky V

Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars

Easy to keep ‘stable’; Casablanca still before Star Wars
RADIX SORT

Radix = “the base of a number system”

- Examples will use base 10 because we are used to that
- In implementations use larger numbers
  - For example, for ASCII strings, might use 128

Idea:

- Bucket sort on one digit at a time
  - Number of buckets = radix
  - Starting with least significant digit
  - Keeping sort stable
- Do one pass per digit
- Invariant: After $k$ passes (digits), the last $k$ digits are sorted
RADIX SORT EXAMPLE

Radix = 10

Input: 478, 537, 9, 721, 3, 38, 143, 67

3 passes (input is 3 digits at max), on each pass, stable sort the input highlighted in yellow

4 7 8
5 3 7
0 0 9
7 2 1
0 0 3
0 3 8
1 4 3
0 6 7

7 2 1
0 0 3
1 4 3
5 3 7
0 6 7
4 7 8
0 3 8
0 0 9

0 0 3
0 0 9
0 3 8
0 6 7
4 7 8
5 3 7
1 4 3
7 2 1
ANALYSIS

Input size: \( n \)
Number of buckets = Radix: \( B \)
Number of passes = “Digits”: \( P \)

Work per pass is 1 bucket sort: \( O(B+n) \)

Total work is \( O(P(B+n)) \)

Compared to comparison sorts, sometimes a win, but often not

- Example: Strings of English letters up to length 15
  - Run-time proportional to: \( 15 \cdot (52 + n) \)
  - This is less than \( n \log n \) only if \( n > 33,000 \)
  - Of course, cross-over point depends on constant factors of the implementations
SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small $n$

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts
SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small $n$

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \log n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
  - Often fastest, but depends on costs of comparisons/copies
SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small $n$

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \log n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
  - Often fastest, but depends on costs of comparisons/copies

$\Omega(n \log n)$ is worst-case and average lower-bound for sorting by comparisons
SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small $n$

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \log n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
  - Often fastest, but depends on costs of comparisons/copies

$\Omega (n \log n)$ is worst-case and average lower-bound for sorting by comparisons

Non-comparison sorts

- Bucket sort good for small number of possible key values
- Radix sort uses fewer buckets and more phases

Best way to sort? It depends!
SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small $n$
- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \log n)$ sorts
- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
  - Often fastest, but depends on costs of comparisons/copies

$\Omega(n \log n)$ is worst-case and average lower-bound for sorting by comparisons

Non-comparison sorts
- Bucket sort good for small number of possible key values
- Radix sort uses fewer buckets and more phases

Best way to sort? It depends!
ALGORITHM DESIGN

- Solving well known problems is great, but how can we use these lessons to approach new problems?
ALGORITHM DESIGN

• Solving well known problems is great, but how can we use these lessons to approach new problems?
  • Guess and Check
ALGORITHM DESIGN

• Solving well known problems is great, but how can we use these lessons to approach new problems?
  • Guess and Check (Brute Force)
ALGORITHM DESIGN

• Solving well known problems is great, but how can we use these lessons to approach new problems?
  • Guess and Check (Brute Force)
  • Linear Solving
ALGORITHM DESIGN

• Solving well known problems is great, but how can we use these lessons to approach new problems?
  • Guess and Check (Brute Force)
  • Linear Solving
  • Divide and Conquer
ALGORITHM DESIGN

- Solving well known problems is great, but how can we use these lessons to approach new problems?
  - Guess and Check (Brute Force)
  - Linear Solving
  - Divide and Conquer
  - Randomization and Approximation
ALGORITHM DESIGN

• Solving well known problems is great, but how can we use these lessons to approach new problems?
  • Guess and Check (Brute Force)
  • Linear Solving
  • Divide and Conquer
  • Randomization and Approximation
  • Dynamic Programming
LINEAR SOLVING

• Basic linear approach to problem solving
LINEAR SOLVING

• Basic linear approach to problem solving
• If the decider creates a set of correct answers, find one at a time
LINEAR SOLVING

• Basic linear approach to problem solving

• If the decider creates a set of correct answers, find one at a time
  • Selection sort: find the lowest element at each run through

• Sometimes, the best solution
  • Find the smallest element of an unsorted array
ALGORITHM DESIGN

• Which approach should be used comes down to how difficult the problem is
ALGORITHM DESIGN

- Which approach should be used comes down to how difficult the problem is
- How do we describe problem difficulty?
  - $P$: Set of problems that can be solved in polynomial time
ALGORITHM DESIGN

• Which approach should be used comes down to how difficult the problem is

• How do we describe problem difficulty?
  • P : Set of problems that can be solved in polynomial time
  • NP : Set of problems that can be verified in polynomial time
ALGORITHM DESIGN

• Which approach should be used comes down to how difficult the problem is

• How do we describe problem difficulty?
  • P : Set of problems that can be solved in polynomial time
  • NP : Set of problems that can be verified in polynomial time
  • EXP: Set of problems that can be solved in exponential time
ALGORITHM DESIGN

• Some problems are provably difficult
ALGORITHM DESIGN

• Some problems are provably difficult
  • Humans haven’t beaten a computer in chess in years, but computers are still far away from “solving” chess
ALGORITHM DESIGN

• Some problems are provably difficult
  • Humans haven’t beaten a computer in chess in years, but computers are still far away from “solving” chess
  • At each move, the computer needs to approximate the best move
Some problems are provably difficult

Humans haven’t beaten a computer in chess in years, but computers are still far away from “solving” chess

At each move, the computer needs to approximate the best move

Certainty always comes at a price
APPROXIMATION DESIGN

• What is approximated in the chess game?
APPROXIMATION DESIGN

• What is approximated in the chess game?
  • Board quality – If you could easily rank which board layout in order of quality, chess is simply choosing the best board
**APPROXIMATION DESIGN**

- **What is approximated in the chess game?**
  - Board quality – If you could easily rank which board layout in order of quality, chess is simply choosing the best board.
  - It is very difficult, branching factor for chess is ~35.
What is approximated in the chess game?

- Board quality – If you could easily rank which board layout in order of quality, chess is simply choosing the best board.
- It is very difficult, branching factor for chess is ~35.
- Look as many moves into the future as time allows to see which move yields the best outcome.
APPROXIMATION DESIGN

- Recognize what piece of information is costly and useful for your algorithm
APPROXIMATION DESIGN

• Recognize what piece of information is costly and useful for your algorithm
  • Consider if there is a cheap way to estimate that information
APPROXIMATION DESIGN

• Recognize what piece of information is costly and useful for your algorithm
  • Consider if there is a cheap way to estimate that information
  • Does your client have a tolerance for error?
  • Can you map this problem to a similar problem?
• “Greedy” algorithms are often approximators
RANDOMIZATION DESIGN

- Randomization is also another approach
RANDOMIZATION DESIGN

• Randomization is also another approach
  • Selecting a random pivot in quicksort gives us more certainty in the runtime
RANDOMIZATION DESIGN

• Randomization is also another approach
  • Selecting a random pivot in quicksort gives us more certainty in the runtime
  • This doesn’t impact correctness, a randomized quicksort still returns a sorted list
RANDOMIZATION DESIGN

• Randomization is also another approach
  • Selecting a random pivot in quicksort gives us more certainty in the runtime
  • This doesn’t impact correctness, a randomized quicksort still returns a sorted list

• Two types of randomized algorithms
  • Las Vegas – correct result in random time
RANDOMIZATION DESIGN

• Randomization is also another approach
  • Selecting a random pivot in quicksort gives us more certainty in the runtime
  • This doesn’t impact correctness, a randomized quicksort still returns a sorted list

• Two types of randomized algorithms
  • Las Vegas – correct result in random time
  • Montecarlo – estimated result in deterministic time
RANDOMIZATION DESIGN

• Can we make a Montecarlo quicksort?
RANDOMIZATION DESIGN

• Can we make a Montecarlo quicksort?
  • Runs $O(n \log n)$ time, but not guaranteed to be correct
RANDOMIZATION DESIGN

• Can we make a Montecarlo quicksort?
  • Runs $O(n \log n)$ time, but not guaranteed to be correct
  • Terminate a random quicksort early!
RANDOMIZATION DESIGN

• Can we make a Montecarlo quicksort?
  • Runs $O(n \log n)$ time, but not guaranteed to be correct
  • Terminate a random quicksort early!
  • If you haven’t gotten the problem in some constrained time, just return what you have.
RANDOMIZATION DESIGN

- How close is a sort?
- If we say a list is 90% sorted, what do we mean?
RANDOMIZATION DESIGN

• How close is a sort?

• If we say a list is 90% sorted, what do we mean?
  • 90% of elements are smaller than the object to the right of it?
RANDOMIZATION DESIGN

• How close is a sort?
• If we say a list is 90% sorted, what do we mean?
  • 90% of elements are smaller than the object to the right of it?
  • The longest sorted subsequence is 90% of the length?
RANDOMIZATION DESIGN

• How close is a sort?

• If we say a list is 90% sorted, what do we mean?
  • 90% of elements are smaller than the object to the right of it?
  • The longest sorted subsequence is 90% of the length?

• Analysis for these problems can be very tricky, but it’s an important approach
ALGORITHMS

• There aren’t many *easy* problems left!
• Understand the tools for problem solving
ALGORITHMS

• There aren’t many easy problems left!
• Understand the tools for problem solving
• Eliminate as many non-feasible solutions as possible
ALGORITHMS

• There aren’t many easy problems left!
• Understand the tools for problem solving
• Eliminate as many non-feasible solutions as possible
• Understand, that some problems are too difficult for a fast, elegant solution
There aren’t many easy problems left!

Understand the tools for problem solving

Eliminate as many non-feasible solutions as possible

Understand, that some problems are too difficult for a fast, elegant solution

Academics are great for providing ideas, but sometimes better asymptotic runtimes don’t become apparent until $n > 10^{10}$
HARDWARE CONSTRAINTS

- So far, we’ve taken for granted that memory access in the computer is constant and easily accessible
HARDWARE CONSTRAINTS

• So far, we’ve taken for granted that memory access in the computer is constant and easily accessible
  • This isn’t always true!
HARDWARE CONSTRAINTS

• So far, we’ve taken for granted that memory access in the computer is constant and easily accessible
  • This isn’t always true!
  • At any given time, some memory might be cheaper and easier to access than others
HARDWARE CONSTRAINTS

• So far, we’ve taken for granted that memory access in the computer is constant and easily accessible
  • This isn’t always true!
  • At any given time, some memory might be cheaper and easier to access than others
  • Memory can’t always be accessed easily
HARDWARE CONSTRAINTS

• So far, we’ve taken for granted that memory access in the computer is constant and easily accessible
  • This isn’t always true!
  • At any given time, some memory might be cheaper and easier to access than others
  • Memory can’t always be accessed easily
  • Sometimes the OS lies, and says an object is “in memory” when it’s actually on the disk
HARDWARE CONSTRAINTS

• Back on 32-bit machines, each program had access to 4GB of memory
HARDWARE CONSTRAINTS

• Back on 32-bit machines, each program had access to 4GB of memory
  • This isn’t feasible to provide!
HARDWARE CONSTRAINTS

• Back on 32-bit machines, each program had access to 4GB of memory
  • This isn’t feasible to provide!
  • Sometimes there isn’t enough available, and so memory that hasn’t been used in a while gets pushed to the disk
HARDWARE CONSTRAINTS

- Back on 32-bit machines, each program had access to 4GB of memory
  - This isn’t feasible to provide!
  - Sometimes there isn’t enough available, and so memory that hasn’t been used in a while gets pushed to the disk
- Memory that is frequently accessed goes to the cache, which is even faster than RAM
The Memory Mountain

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 cache

Ridges of Temporal Locality

Slopes of Spatial Locality

read throughput (MB/s)

stride (words)

working set size (bytes)
NEXT WEEK

• No class on Monday – Happy Memorial Day!
NEXT WEEK

• No class on Monday – Happy Memorial Day!

• Formalize discussion of the “memory mountain” and how this should impact your design decisions