Objectives

- Finish survey of common running times
- More on Data structures

- Checking in on journal
  - Alternative to quizzes

A SURVEY OF COMMON RUNNING TIMES

Review: O(n) Algorithms

- Constant work on each input element
- Examples:
  - Finding the max
  - Merging two sorted lists

O(n log n) Time

- Also referred to as ***linearithmic*** time
- Arises in divide-and-conquer algorithms
  - Splitting input into equal pieces, solve recursively, combine solutions in linear time

What well-known set of algorithms has an O(n logn) running time?

O(n log n) Time Example

- Sorting: Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons
- Mergesort
  1. Break input into equal-sized pieces
  2. Sorts each half recursively
  3. Merges sorted halves into a sorted list

Talk about the bound on running time during D&C chapter…

O(n log n) Time Example

- Largest empty interval. Given *n* (not necessarily ordered) time-stamps $x_1, \ldots, x_n$ at which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) Time Example

- Largest empty interval. Given *n* (not necessarily ordered) time-stamps $x_1, \ldots, x_n$ at which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?
- O(n log n) solution
  1. Sort time-stamps
  2. Scan sorted list in order, identifying the maximum gap between successive time-stamps

Quadratic Time: O(n^2)

- Examples?

Quadratic Time: O(n^2)

- Examples:
  - Enumerate all pairs of elements
  - Often involves nested loops (n iterations each)

Quadratic Time: O(n^2)

- Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest
- O(n^2) solution. Try all pairs of points

```
min = (x1 - x2)² + (y1 - y2)²
for i = 1 to n
   for j = i+1 to n
      d = (xi - xj)² + (yi - yj)²
      if (d < min)
         min = d
```

don't need to take square roots

$\Omega(n^2)$ seems inevitable, but Chapter 5 has an O(n logn) solution

Cubic Time: O(n^3)

- Examples?

Cubic Time: O(n^3)

- Enumerate all triples of elements

Cubic Time: $O(n^3)$

- Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?
- $O(n^3)$ solution. For each pair of sets, determine if they are disjoint

```
foreach set Si
   foreach other set Sj
      foreach element p of Si
         determine whether p also belongs to Sj

      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
```


Polynomial Time: $O(n^k)$ Time

- To get all pairs, the algorithm is $O(n^2)$

What is an example of an $O(n^k)$ algorithm?

All subsets of size *k*

Polynomial Time: $O(n^k)$ Time

- Independent set of size *k*. Given a graph, are there *k* nodes such that no two are joined by an edge?
  - *k* is a constant

Polynomial Time: $O(n^k)$ Time

- Independent set of size *k*. Given a graph, are there *k* nodes such that no two are joined by an edge?
  - *k* is a constant

```
foreach subset S of k nodes
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
```

- $O(n^k)$ solution
  1. Enumerate all subsets of k nodes

     $$\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \le \frac{n^k}{k!}$$

  2. Check whether S is an independent set = $O(k^2)$.

$O(k^2\, n^k / k!) = O(n^k)$

poly-time for k=17 but not practical

Exponential Time

- Independent set. Given a graph, what is the *maximum size* of an independent set?
- $O(n^2\, 2^n)$ solution. Enumerate all subsets

```
S* = φ
foreach subset S of nodes
   check whether S in an independent set
   if (S is largest independent set seen so far)
      S* = S
```


O(log n) Time

- ***Sublinear*** time
- Know any algorithms that take O(log n) time?

O(log n) Time

- Example: Binary search
- Often requires some pre-processing or data structure that allows cheaper "querying" than *n* time

Summary of Running Times

Running Time	Example
O(log n)	Generally dividing problem in half on each iteration
O(n)	Operate on each input value
O(n log n)	Divide and conquer
O(n^2)	Operate on each pair of inputs
O(n!)	Operate on each permutation of inputs

MORE COMPLEX DATA STRUCTURES

Improving Running Times

After overcoming higher-level obstacles, lower-level **implementation details** can **improve runtime**.

PRIORITY QUEUES

Priority Queues

- Elements have a ***priority*** or *key*
- Each time select an element from the priority queue, want the one with *highest* priority
- More formally…
  - Maintains a set of elements *S*
    - Each element $v \in S$ has a key(v) for its priority
      - Smaller keys represent higher priorities
  - Supported operations
    - Add, delete elements
    - Select element with smallest key

Key	2	4	5	6	9	20
Value	3542	5143	8712	1264	9123	5954

Priority
Process id

Not implementation, just how to envision

Motivating Example: Scheduling Processes

Key	2	4	5	6	9	20
Value	3542	5143	8712	1264	9123	5954

Priority
Process id

- Each process has a priority or urgency
- Processes do not arrive in priority order
- **Goal**: run process with highest priority

Using a Priority Queue

How could we use a PQ to sort a list of numbers?

Priority Queues for Sorting

1. Add elements into PQ with the number's value as its priority
2. Then extract the smallest number *until* done
   - Come out in sorted order

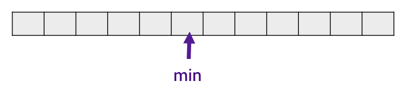
Sorting *n* numbers takes O(n logn) time

What is the goal running time for our PQ's operations? **O(logn)**

Already know our "loops" will be O(n)

Implementing a Priority Queue

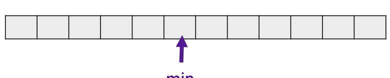
- Consider an *unordered* list, where there is a pointer to minimum



- How difficult (i.e., expensive) is
  - Adding new elements?
  - Extraction?

Implementing a Priority Queue

- Consider an *unordered* list, where there is a pointer to minimum



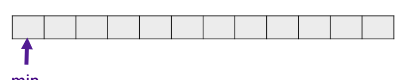
- How difficult (i.e., expensive) is
  - Adding new elements? *easy*
  - Extraction? *difficult*
    - Need to find "new" minimum: O(n)

What is the running time for sorting with the PQ in this case?

$O(n^2)$

Implementing a Priority Queue

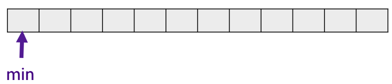
- Consider a *sorted* list where min is at the beginning



- Should you use an array or linked list?
- How difficult is
  - Adding new elements?
  - Extraction?

Implementing a Priority Queue

- Consider a sorted list where min is at the beginning



- Should you use an array or linked list?
- How difficult is
  - Adding new elements? *more difficult (insertion)*
  - Extraction? *Easy*

What is the running time for sorting with the PQ in this case?

$O(n^2)$

Reflection

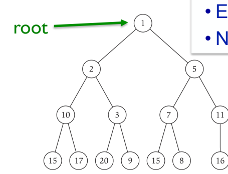
- All of "known" data structures has one operation that takes O(n) time
- Cannot implement PQs with "known" data structures arrays and lists to meet desired O(n log n) runtime

Motivates use of a new data structure (***heap)*** to implement PQ

HEAPS

Heap Defined

- Combines benefits of sorted array and list
- Balanced binary tree

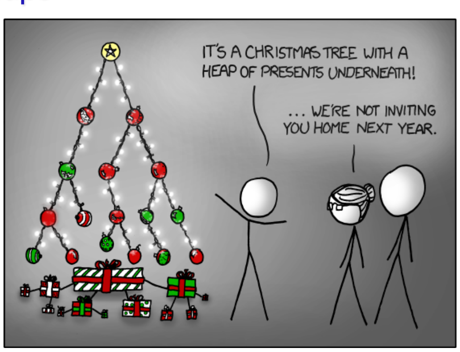


- Each node has *at most* 2 children
- Node value is its *key*

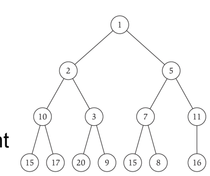
Heap order: each node's key is at least as large as its parent's

Note: **not** a binary search tree

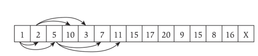
Heaps



Implementing a Heap



- Option 1: Use pointers
  - Each node keeps
    - Element it stores (key)
    - 3 pointers: 2 children, parent
- Option 2: No pointers
  - Requires knowing upper bound on *n*
  - For node at position *i*
    - left child is at *2i*
    - right child is at *2i+1*



If know child's position, what is the position of parent?

Assignment

- Problem Set Due Friday
- Finish reading, summarizing Chapter 2

Jan 19, 2011 Sprenkle - CSCI211 37