Lecture 7:
Heapsort / Priority Queues

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.stonybrook.edu/~skiena

Problem of the Day

Take as input a sequence of $2n$ real numbers. Design an $O(n \log n)$ algorithm that partitions the numbers into n pairs, with the property that the partition minimizes the maximum sum of a pair.

For example, say we are given the numbers (1,3,5,9). The possible partitions are ((1,3),(5,9)), ((1,5),(3,9)), and ((1,9),(3,5)). The pair sums for these partitions are (4,14), (6,12), and (10,8). Thus the third partition has 10 as its maximum sum, which is the minimum over the three partitions.

Importance of Sorting

Why don't CS profs ever stop talking about sorting?

- Computers spend a lot of time sorting, historically 25% on mainframes.
- Sorting is the best studied problem in computer science, with many different algorithms known.
- Most of the interesting ideas we will encounter in the course can be taught in the context of sorting, such as divide-and-conquer, randomized algorithms, and lower bounds.

You should have seen most of the algorithms, so we will concentrate on the analysis.

Efficiency of Sorting

Sorting is important because that once a set of items is sorted, many other problems become easy.

Further, using $O(n \log n)$ sorting algorithms leads naturally to sub-quadratic algorithms for all these problems.

n	$n^2/4$	$n \lg n$
10	*25*	33
100	2,500	*664*
1,000	250,000	*9,965*
10,000	25,000,000	*132,877*
100,000	2,500,000,000	*1,660,960*
1,000,000	250,000,000,000	*13,815,551*

Large-scale data processing is impossible with $\Omega(n^2)$ sorting.

Application of Sorting: Searching

Binary search lets you test whether an item is in a dictionary in $O(\lg n)$ time.

Search preprocessing is perhaps the single most important application of sorting.

Application of Sorting: Closest pair

Given n numbers, find the pair which are closest to each other.

Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an $O(n)$ linear scan completes the job.

Application of Sorting: Element Uniqueness

Given a set of n items, are they all unique or are there any duplicates?

Sort them and do a linear scan to check all adjacent pairs.

This is a special case of closest pair above.

Application of Sorting: Mode

Given a set of n items, which element occurs the largest number of times? More generally, compute the frequency distribution.

Sort them and do a linear scan to measure the length of all adjacent runs.

The number of instances of k in a sorted array can be found in $O(\log n)$ time by using binary search to look for the positions of both $k - \epsilon$ and $k + \epsilon$.

Application of Sorting: Median and Selection

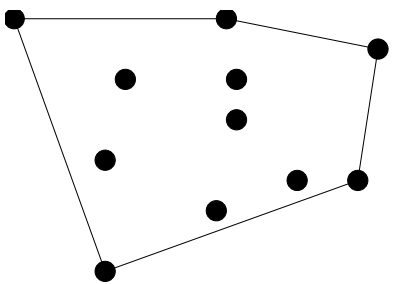
What is the kth largest item in the set?

Once the keys are placed in sorted order in an array, the kth largest can be found in constant time by simply looking in the kth position of the array.

There is a linear time algorithm for this problem, but the idea comes from partial sorting.

Application of Sorting: Convex hulls

Given n points in two dimensions, find the smallest area polygon which contains them all.

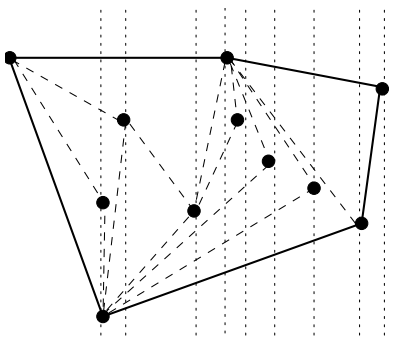


The convex hull is like a rubber band stretched over the points.

Convex hulls are the most important building block for more sophisticated geometric algorithms.

Finding Convex Hulls

Once you have the points sorted by x-coordinate, they can be inserted from left to right into the hull, since the rightmost point is always on the boundary.



Sorting eliminates the need check whether points are inside the current hull.

Adding a new point might cause others to be deleted.

Pragmatics of Sorting: Comparison Functions

Alphabetizing is the sorting of text strings.
Libraries have very complete and complicated rules concerning the relative *collating sequence* of characters and punctuation.

- Is *Skiena* the same key as *skiena*?
- Is *Brown-Williams* before or after *Brown America* before or after *Brown, John*?

Explicitly controlling the order of keys is the job of the *comparison function* we apply to each pair of elements, including the question of increasing or decreasing order.

Pragmatics of Sorting: Equal Elements

Elements with equal keys will all bunch together in any total order, but sometimes the relative order among these keys matters.

Often there are secondary keys (like first names) to test after the primary keys. This is a job for the comparison function.

Certain algorithms (like quicksort) require special care to run efficiently with large numbers of equal elements.

Pragmatics of Sorting: Library Functions

Any reasonable programming language has a built-in sort routine as a library function.

You are almost always better off using the system sort than writing your own routine.

For example, the standard library for C contains the function `qsort` for sorting:

```
void qsort(void *base, size_t nel, size_t width,
        int (*compare) (const void *, const void *));
```

Selection Sort

Selection sort scans throught the entire array, repeatedly finding the smallest remaining element.

For $i = 1$ to n

A: Find the smallest of the first $n - i + 1$ items.

B: Pull it out of the array and put it first.

Selection sort takes $O(n(T(A) + T(B))$ time.

The Data Structure Matters

Using arrays or unsorted linked lists as the data structure, operation A takes $O(n)$ time and operation B takes $O(1)$, for an $O(n^2)$ selection sort.

Using balanced search trees or heaps, both of these operations can be done within $O(\lg n)$ time, for an $O(n \log n)$ selection sort called heapsort.

Balancing the work between the operations achieves a better tradeoff.

Key question: "Can we use a different data structure?"

Priority Queues

Priority queues are data structures which provide extra flexibility over sorting.

This is important because jobs often enter a system at arbitrary intervals. It is more cost-effective to insert a new job into a priority queue than to re-sort everything on each new arrival.

Priority Queue Operations

The basic priority queue supports three primary operations:

- *Insert(Q,x)*: Given an item x with key k, insert it into the priority queue Q.
- *Find-Minimum(Q)* or *Find-Maximum(Q)*: Return a pointer to the item whose key is smaller (larger) than any other key in the priority queue Q.
- *Delete-Minimum(Q)* or *Delete-Maximum(Q)* – Remove the item from the priority queue Q whose key is minimum (maximum).

Each of these operations can be easily supported using heaps or balanced binary trees in $O(\log n)$.

Applications of Priority Queues: Dating

What data structure should be used to suggest who to ask out next for a date?

It needs to support retrieval by *desirability*, not name.

Desirability changes (up or down), so you can re-insert the max with the new score after each date.

New people you meet get inserted with your observed desirability level.

There is no reason to delete anyone until they rise to the top.

Applications of Priority Queues: Discrete Event Simulations

In simulations of airports, parking lots, and jai-alai – priority queues can be used to maintain who goes next.
The stack and queue orders are just special cases of orderings.
In real life, certain people cut in line, and this can be modeled with a priority queue.

Heap Definition

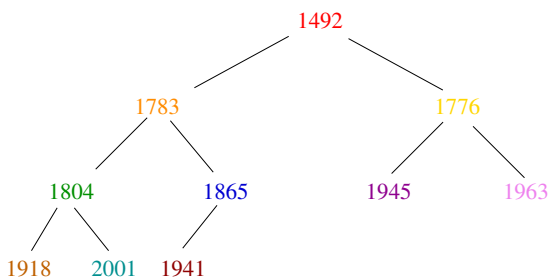
A *binary heap* is defined to be a binary tree with a key in each node such that:

1. All leaves are on, at most, two adjacent levels.
2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
3. The key in root is $\leq$ all its children, and the left and right subtrees are again binary heaps.

Conditions 1 and 2 specify shape of the tree, and condition 3 the labeling of the tree.

Binary Heaps

Heaps maintain a partial order on the set of elements which is weaker than the sorted order (so it can be efficient to maintain) yet stronger than random order (so the minimum element can be quickly identified).



1	1492
2	1783
3	1776
4	1804
5	1865
6	1945
7	1963
8	1918
9	2001
10	1941

A heap-labeled tree of important years (l), with the corresponding implicit heap representation (r)

Heapsort Animation

Watch as

- We build the heap by repeated insertion.
- Embed it in an array.
- Then repeatedly remove the maximum to sort:

https://upload.wikimedia.org/wikipedia/commons/4/4d/Heapsort-example.gif

The partial order defined by the heap structure is weaker than sorting, which explains why it is easier to build: linear time if you do it right.

Array-Based Heaps

The most natural representation of this binary tree would involve storing each key in a node with pointers to its two children.

However, we can store a tree as an array of keys, using the position of the keys to *implicitly* satisfy the role of the pointers.

The *left* child of k sits in position $2k$ and the right child in $2k + 1$.

The parent of k is in position $\lfloor n/2 \rfloor$.

Can we Implicitly Represent Any Binary Tree?

The implicit representation is only efficient if the tree is sparse, meaning that the number of nodes $n < 2^h$.

All missing internal nodes still take up space in our structure.

This is why we insist on heaps as being as balanced/full at each level as possible.

The array-based representation is also not as flexible to arbitrary modifications as a pointer-based tree.

Constructing Heaps

Heaps can be constructed incrementally, by inserting new elements into the left-most open spot in the array.
If the new element is greater than its parent, swap their positions and recur.
Since all but the last level is always filled, the height h of an n element heap is bounded because:

$$\sum_{i=1}^{h} 2^i = 2^{h+1} - 1 \geq n$$

so $h = \lfloor \lg n \rfloor$.
Doing n such insertions really takes $\Theta(n \log n)$, because the last $n/2$ insertions require $O(\log n)$ time each.

Heap Insertion

```
pq_insert(priority_queue *q, item_type x)
{
        if (q->n >= PQ_SIZE)
                printf("Warning: overflow insert");
        else {
                q->n = (q->n) + 1;
                q->q[ q->n ] = x;
                bubble_up(q, q->n);
        }
}
```

Bubble Up

```
bubble_up(priority_queue *q, int p)
{
        if (pq_parent(p) == -1) return;

        if (q->q[pq_parent(p)] > q->q[p]) {
                pq_swap(q,p,pq_parent(p));
                bubble_up(q, pq_parent(p));
        }
}
```

An Even Faster Way to Build a Heap

Given two heaps and a fresh element, they can be merged into one by making the new one the root and bubbling down (*heapify*).

Build-heap(A)

$n = |A|$

For $i = \lfloor n/2 \rfloor$ to 1 do

Heapify(A,i)

Bubble Down Implementation

```
bubble_down(priority_queue *q, int p)
{
        int c; (* child index *)
        int i; (* counter *)
        int min_index; (* index of lightest child *)

        c = pq_young_child(p);
        min_index = p;

        for (i=0; i<=1; i++)
                if ((c+i) <= q->n) {
                        if (q->q[min_index] > q->q[c+i]) min_index = c+i;
                }

        if (min_index ! = p) {
        pq_swap(q,p,min_index);
        bubble_down(q, min_index);
        }
}
```

Exact Analysis of Heapify

In fact, build-heap performs better than $O(n \log n)$, because most of the heaps we merge are extremely small.

It follows the same analysis as dynamic arrays (Chapter 3). In a full binary tree on n nodes, there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h, so the cost of building a heap is:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h)$$

Since this sum is not quite a geometric series, we can't apply the usual identity to get the sum. But it should be clear that the series converges.

Proof of Convergence (*)

The identify for the sum of a geometric series is

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

If we take the derivative of both sides, ...

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

Multiplying both sides of the equation by x gives:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Substituting $x = 1/2$ gives a sum of 2, so Build-heap uses at most $2n$ comparisons and thus linear time.

Is our Analysis Tight?

"Are we doing a careful analysis? Might our algorithm be faster than it seems?"

Doing at most x operations of at most y time each takes total time $O(xy)$. But, if we overestimate too much, our bound may not be as tight!

Heapsort

Heapify can be used to construct a heap, using the observation that an isolated element forms a heap of size 1.

Heapsort(A)

    Build-heap(A)
    for $i = n$ to 1 do
        swap(A[1],A[i])
        $n = n - 1$
        Heapify(A,1)

Exchanging the maximum element with the last element and calling heapify repeatedly gives an $O(n \lg n)$ sorting algorithm. Why is it not $O(n)$?