Changing the Model

What if we settle for the “approximate best?”

Types of guarentees, given that the algorithm produces X and the best is Y:

1. $X = Y$.
2. X's rank is “close to” Y's rank:

$$rank(X) \leq rank(Y) + \text{“small”}.$$

3. X is “usually” Y.

$$\mathbf{P}(X = Y) \geq \text{“large”}.$$

4. X's rank is “usually” “close” to Y's rank.

We often give such algorithms names:

1. Exact or deterministic algorithm.
2. Approximation algorithm.
3. Probabilistic algorithm.
4. Heuristic.

We can also sacrifice reliability for speed:

1. We find the best, “usually” fast.
2. We find the best fast, or we don't get an answer at all (but fast).

Examples for Findmax

Choose m elements at random, and pick the best.

- For large n, if $m = \log n$, the answer is pretty good.
- Cost is $m - 1$.
- Rank is $\frac{mn}{m+1}$.

Probabilistic Algorithms

Probabilistic algorithms include steps that are affected by **random** events.

Problem: Pick one number in the upper half of the values in a set.

1. Pick maximum: $n-1$ comparisons.
2. Pick maximum from just over 1/2 of the elements: $n/2$ comparisons.

Can we do better? Not if we want a **guarantee**.

Probabilistic Algorithm

Pick 2 numbers and choose the greater.

This will be in the upper half with probability 3/4.

Not good enough? Pick more numbers!

For k numbers, greatest is in upper half with probability $1 - 2^{-k}$.

Monte Carlo Algorithm: Good running time, result not guaranteed.

Las Vegas Algorithm: Result guaranteed, but not the running time.

Sorting

Initial model:

- Sort key has a linear order (comparable).
- We have an array of elements.
- We wish to sort the elements in the array.
- We get information about elements only by comparison of two elements.
- We can preserve order information only by swapping a pair of elements.

To simplify analysis:

- Assume all elements are unique.
- For analysis purposes, consider the input to be a permutation of the values 1 to n.

What if the ALGORITHM could make this assumption?

Swap Sorts

Repeatedly scan input, swapping any out-of-order elements.

Bubble sort: $O(n^2)$ in worst case.

Inversions of an element: the number of smaller elements to the right of the element.

The sum of inversions for all elements is the number of swaps required by bubblesort.

ANY algorithm that removes one inversion per swap requires at least this many swaps.

Worst number of inversions:

Best number of inversions:

Average number of inversions:

- Note that the sum of the total inversions for any permutation and its reverse is $\frac{n(n-1)}{2}$.
- Alternative view: every one of the $\frac{n(n-1)}{2}$ possible inversions occurs in a given permutation or its reverse.

Heap Sort

Heap: complete binary tree with the value of any node at least as large as its two children.

Algorithm:

- Build the heap.
- Repeat n times:
  - Remove the root.
  - Repair the heap.

This gives us list in reverse sorted order.

Since the heap is a complete binary tree, it can be stored in an array.

To delete max element:

- Swap the last element in the heap with the first (root).
- Repeatedly swap the placeholder with larger of its two children until done.

Building the heap

To build a heap, first heapify the two subheaps, then push down the root to its proper position.

Cost:

$$f(n) \leq 2f(n/2) + 2\log n.$$

Alternatively: Start at first internal node and, moving up the array, siftdown each element.

Cost:

$$\begin{aligned} f(n) &= \sum_{i=1}^{\log n} (i-1)\frac{n}{2^i} \\ &= \frac{n}{2}\sum_{i=1}^{\log n - 1} \frac{i}{2^i} \\ &< 2\frac{n}{2} = n. \end{aligned}$$

Quicksort

Algorithm:

- Pick a pivot value.
- Split the array into elements less than the pivot and elements greater than the pivot.
- Recursively sort the sublists.

Worst case:

Pick the pivot at random, so that no particular input has bad performance.

Quicksort Average Cost

$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(f(i) + f(n-i-1)) & n > 1 \end{cases}$$

Since the two halves of the summation are identical,

$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} f(i) & n > 1 \end{cases}$$

Multiplying both sides by n yields

$$nf(n) = n(n-1) + 2\sum_{i=0}^{n-1} f(i).$$

Average Cost (cont.)

Get rid of the full history by subtracting $nf(n)$ from $(n+1)f(n+1)$

$$\begin{aligned}
nf(n) &= n(n-1) + 2\sum_{i=1}^{n-1} f(i) \\
(n+1)f(n+1) &= (n+1)n + 2\sum_{i=1}^{n} f(i) \\
(n+1)f(n+1) - nf(n) &= 2n + 2f(n) \\
(n+1)f(n+1) &= 2n + (n+2)f(n) \\
f(n+1) &= \frac{2n}{n+1} + \frac{n+2}{n+1}f(n).
\end{aligned}$$

Average Cost (cont.)

Note that $\frac{2n}{n+1} \leq 2$ for $n \geq 1$. Expanding the recurrence, we get

$$\begin{aligned}
f(n{+}1) &\leq 2 + \frac{n+2}{n+1} f(n) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n} f(n-1)\right) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}\left(2 + \frac{n}{n-1} f(n-2)\right)\right) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \cdots + \frac{4}{3}(2 + \frac{3}{2} f(1))\right) \\
&= 2\left(1 + \frac{n+2}{n+1} + \frac{n+2}{n+1}\frac{n+1}{n} + \cdots \right. \\
&\qquad \left. + \frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right) \\
&= 2\left(1 + (n+2)\left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2}\right)\right) \\
&= 2 + 2(n+2)\left(\mathcal{H}_{n+1} - 1\right) \\
&= \Theta(n \log n).
\end{aligned}$$

Lower Bound for Sorting

What is the smallest number of comparisons needed to sort n values?

Clearly, sorting is as hard as finding the min and max element: $\lceil 3n/2 \rceil - 2$.

- Why?

<u>**Information theory**</u> says that, if an algorithm uses only binary decisions to distinguish between n possibilities, then it must use at least $\log n$ such decisions on average.

How is this relevant?

There are $n!$ permutations to the input array.

So, by information theory, we need at least $\log n! = \Theta(n \log n)$ comparisons.

Using the decision tree model, what is the average depth of a node?

This is also $\Theta(\log n!)$.

Linear Insert Sort

Put the element i into a sorted list of the first $i-1$ elements.

Worst case cost:

Best case cost:

Average case cost:

What if we use binary search? (This is called binary insert sort.)

Optimal Sorting

If we count ONLY comparisons, binary insert sort is pretty good.

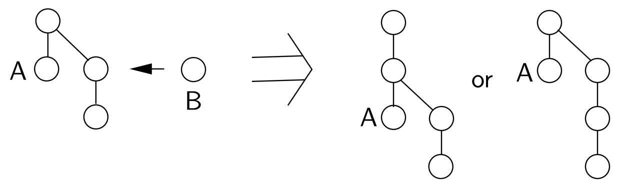
What is the absolute minimum number of comparisons needed to sort?

For $n = 5$, how many comparisons do we need for binary insert sort?

Binary search is best for what values of n?

Binary search is worst for what values of n?

Build the following poset:



Now, put in the fifth element into the chain of 3.

Now, put in the off-element.

Total cost?

Ten Elements

Pair the elements: 5 comparisons.

Sort the winners of the pairings, using the previous algorithm: 7 comparisons.

Now, all we need to do is to deal with the original losers.

General algorithm:

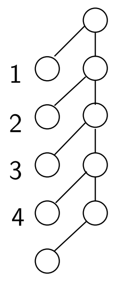
- Pair up all the nodes with $\lfloor \frac{n}{2} \rfloor$ comparisons.
- Recursively sort the winners.
- Fold in the losers.

Finishing the Sort

We will use binary insert to place the losers.

However, we are free to choose the best ordering for inserting.

Recall that binary search is best for $2^k - 1$ items.



Pick the order of inserts to optimize the binary searches.

- 3 (2 compares: size 3)
- 4 (2 compares: size 3)
- 1 (3 compares: size 7)
- 2 (3 compares: size 7)

We can form an algorithm: Binary Merge.

This sort is called **merge insert sort**

Optimal Sort Algorithm?

Merge insert sort is pretty good, but is it optimal?

It does not match the information theoretic lower bound for $n = 12$.

- Merge insert sort gives 30 instead of 29 comparison.

BUT, exhaustive search shows that the information theoretic bound is an underestimate for $n = 12$. 30 is best.

Call the optimal worst cost for n elements $S(n)$.

- $S(n+1) \leq S(n) + \lceil \log(n+1) \rceil$.
  Otherwise, we would sort n elements and binary insert the last.
- For all n and m,
  $S(n+m) \leq S(n) + S(m) + M(m,n)$ for $M(m,n)$ the best time to merge two sorted lists.
- For $n = 47$, we can do better by splitting into pieces of size 5 and 42, then merging.

A Truly Optimal Algorithm

Pick the best set of comparisons for size 2.

Then for size 3, 4, 5, ...

Combine them together into one program with a big case statement.

Is this an algorithm?