Sorting-Searching

June 19, 2020

1 Applications of Sorting

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*

The table above shows that clever sorting algorithms exist that run in $ O(n log n)$ and naive algorithms run in $ O(n^2) $

1.1 Uses of Sorting

1.1.1 Searching

- Binary search tests whether an item is in a dictionary in $ O(log n) $ time, provided the keys are all sorted. Preprocessing is perhpas the single most important application of sorting

1.1.2 Closest Pair

- Ginven a set of $ n $ numbers, how do we find the pair of numbers that have the smallest difference between them? Once the numbers are sorted, the closest pair of numbers must lie next to each other somewhere in sorted order. Thus, a linear-time scan through them completes the job, for a total of $ O (n log n) $ time including the sorting

1.1.3 Element Uniqueness

- Are there any duplicates in a given set of $ n $ items? This is a special case of the closest-pair problem above, where we ask if there is a pair separated by a gap of zero. The most efficient algortihm sorts the numbers and then does a linear scan through checking all adjacent pairs

1.1.4 Frequency Distribution

- Given a set of $ n $ items, which element occurs the largest number of times in the set? If the items are sorted, we can sweep from left to right and count them, since all identical items will be lumped together during sorting
- To find out how often an arbitrary element $ k $ occurs, look up $ k $ using binary search in a sorted array of keys. By walking to the right, we can count in $ O(\log n + c) $ time, where c is the number of occurences of $ k $. Even better, the number of instances of $ k $ can be found in $ O (\log n) $ time by using binary search to look for the position of both $ K - dx $ and $ k + dx $ where dx is arbitrarily small, and then taking the difference of these positions

1.1.5 Selection

- What is the $ kth $ largest item in an array? If the keys are placed in sorted order, the $ kth $ largest can be found in constant time by simply looking at $ kth $ position of the array. In particular, the median element appears in the $ (n/2)$ position in sorted order

1.1.6 Conver Hulls

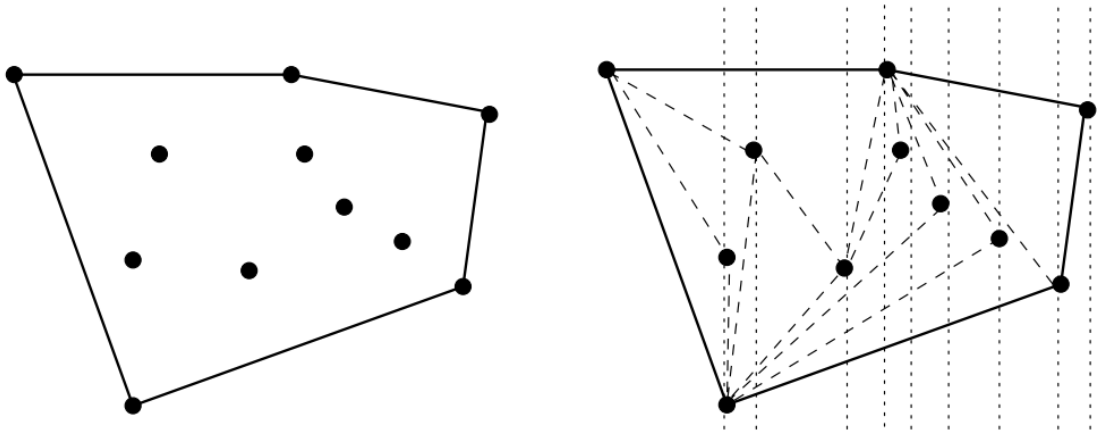


Figure 1: The convex hull of a set of points (1), constructed by left-to-right insertion. Think of it like a rubber band around toothpicks

What is the polyogn of smallest area that contains a given set of $ n $ points in two dimensions?

Once you have the points sorted by x-coordinate, the points can be inserted from left to right into the hull. Since the right-most point is always the boundary, we know that will appear in the hull. Adding this new right-most point may cause others to be deleted, but we can quickly identify these points because they lie inside the polygon formed by adding the new points. These points will be neighbors of the previous point we inserted, so they will be easy to find and delete. The total time is linear after the sorting has been done

Generally speaking, never be afraid to spend time sorting, provided you use an efficient sorting routine

1.2 Finding the Intersection

1.2.1 Problem:

- Given an efficent algorithm to determine whether two sets (of size $ m $ and $ n $, respectively) are disjoint. Analyze the worst-case complexity in terms of $ m $ and $ n $ considering the case where $ m $ is substantially smaller than $ n $

1.2.2 Solutions

First sort the big set : the big set can be sorted in $ O (n log n) $ times. We can now do a binary search with each of the $ m $ elements in the second, looking to see if it exists in the big set. The total time will be $ O ((n + m) log n) $

first sort the small set : The small set can be sorted in $ O (m log m) $ time. We can now do a binary search with each of the $ n $ elements in the big set, looking ot see if it exists in the small one. The total time will be $ O ((n + m) log m) $

Sort both sets : Observe that once the two sets are sorted, we no longer have to do binary search to detect a common element. W can just compare the smallest of the two sorted sets, and discard the smaller one if they are not identical. By repeating this idea recursively on the now smaller sets, we can test for duplication in linear time after sorting. The total cost is $ O (n log n + m log m + n + m)$

1.2.3 Answer

the small-set sorting tumps bit-set sorting since $ log m < log n $ when $ m < n $. Thus sorting the small set is the best of these options. Note that this is linear when $ m $ is constant in size.

Furthermore that expected linear time can be avhived by hashing. Build a hash table containing the elements of both sets, and verify that collisions in the same bucket are in fact identical elements

2 Pragmatic of Sorting

What order do we want to sort items in?

- **Increasing** or **Decreasing**:
  - A set of keys $ S $ are sorted in ascending order when $ S_i <= S_{i+1} $ for all $ 1 <= i <= n $. They are in descending order when $ S_i >= S_{i+1} $ for all $ 1 <= i < n $. Different applications call for different order
- Sort **key** or entire **record**:
  - Sorting a data set involved maintaining the integrity of complex data records. A mailing list of names, adresses, and phone numbers may be sorted by name as the key field, but it had better retain the linkage between names and adresses. Thus we need to specify which feild is the key feild in our complex record and understand the full extent of each record
- What should we do with equal keys:
  - Sometimes the relative order of the key matter. What if two names are the same, Micheal Jordan. Well you might want to sort according to popularity and it matters which one comes first. It might be required to leave the items in the same relative order as in the

orginal permutation. Sorting algorithms that automatically enforce this requirement are called *stable*. Fast algortihms are generally not stable. **Stability can be achived for any sorting algortihm by adding the inital position as a secondary key**

- What about **non-numerical** data:
  - Alphabetizing the sorting text strings. Libraries have very complete and complicated rules concerning the relative *collating sequence* of characters and punctuations.

3 Heapsort: Fast Sorting via Data Structures

3.1 Selection Sort

Selection sort repeatedly extracts the smallest remaining element from the unsorted part of the set

```
SelectionSort(A)
    For i = 1 to n do
        Sort[i] = Find-Minimum from A
        Delete-Minimum from A
    Return(Sort)
```

Selection sort performs $ n $ iterations, where the average iteration takes $ n / 2 $ steps for a tital of $ O(n^2)$ time

It takes $ O (1) $ time to remove a particular item fom an unsorted array once it has been located, but $ O(n) $ time to find the smallest item. This is essentially what a priority queue does

What happens if we replace the data structute with a better priority queue implementation, either a heap or a balanced binary tree. Operations within the loop now take $ O (log n) $ time each instead of $ O (n) $. Using such a priority queue implementation speeds up selection sort from $ O(n^2) $ to $ O(n log n) $

The name typically given to this algorithm, *heapsort* , obscures the relationsship betwen them, but heapsort is nothing but an implementation of selection sort using the right data structure

3.2 Heaps

More on Python Heaps: https://www.educative.io/edpresso/what-is-the-python-priority-queue

Heaps are a data structure for effciently supporting the priority queue operations insert and extract-min. They work by maintaining a partial order on the set of elements which is weaker than the sorted order (so it can be efficent to maintain) yet stronger than random order (so the minimum element can be quickly identified)

Power in any hierarchically-structured organization is reflected by a tree, where each node in the tree represents a person, and $ edge (x, y) $ implies that $ x $ directly supervises (or dominates) $ y $. The fellow at the root sits at the “top of the heap”

In this way, a *heap-labled tree* is defined to be a binary tree such that the key labeling of each node *dominates* the key labeling of each of its children.

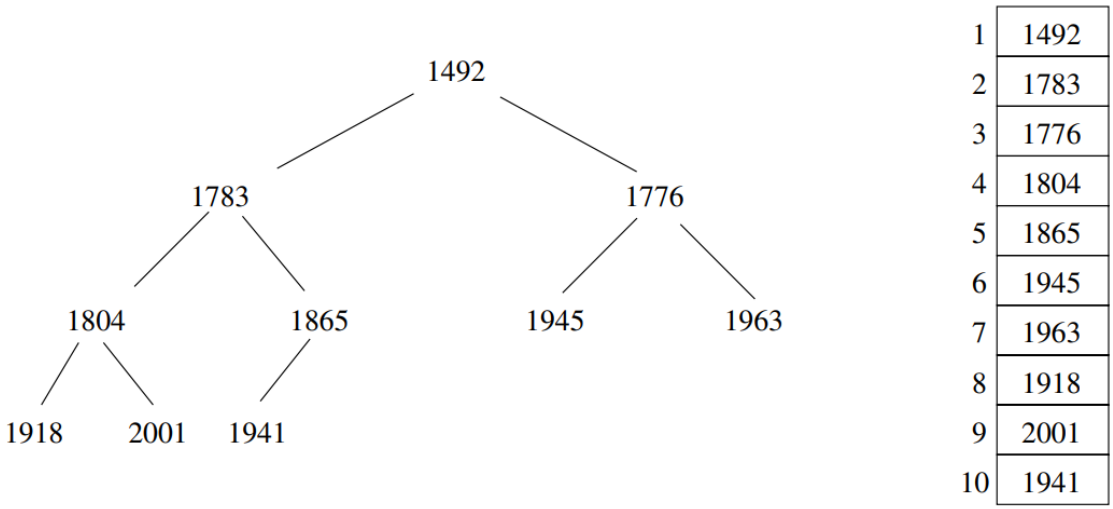


Figure 2: A heap-labeled tree of important years American history $ (1) $, with the corresponding implicit heap representation $(r) $

In a min-heap, a node dominates its children by containing a smaller key than they do, while in a max-heap, parent nodes dominate by being bigger

The most natural implementation of this binary tree would store each key in a node with pointers to its two children. As with binary search trees, the memory used by the pointers can easily outweight the size of the keys, which is the data we are intrested in

The heap is a slick data structure that enables us to represent binary trees without using any pointers. We will store the root of the tree in the first position, respectively. In general, we will store $ 2^l $ keys of the $ lth $ level of a complete binary tree from left to right in positions $ 2^{l-1) $ to $ 2^l -1$ as shown in **Figure 2(r)**

```
typedef struct {
    item_type q[PQ_SIZE+1]; /* body of queue */
    int n; /* number of queue elements */
} priority_queue;
```

What is espically nice about this representation is that the position of the parent and children of the key at position $ k $ are readily determined. The left child of $ k $ sits in position $ 2k $ and the right child in $ 2k + 1 $, while the parent of $ k $ holds court in position $ [n/2] $. This we can move around the tree without pointers

```
pq_parent(int n)
{
    if (n == 1) return(-1);
    else return((int) n/2); /* implicitly take floor(n/2) */
}
pq_young_child(int n)
{
return(2 * n);
}
```

So what the catch with this, representing a binary tree without pointers? Well all internal nodes

still take up space in our structure, since we must represent a full binary tree to maintain the positional mapping between parents and children

Space efficency thus demands that we not allow holes in out tree, that each level be packed as much as it can be

Only the last level may be incomplte. By packing the elements of the last level as far to the left as possible, we can represent an $ n$-key tree using exactly $ n $ elements of the array. If we do not enforce this structural constraints, we might need an array of size $ 2^n $ to store the same elements

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \geq n$$

$$\text{so } h = \lfloor \lg n \rfloor .$$

Since all but the last level is always filled, the height $ h $ of an $ n $ element heap is logarithmic

Loss of flexiblity means we cannot use this idea to represent a binary search tree but works for heaps

- **We cannto efficiently search for a particular key in a heap. Binary search does not work because a heap is not a binary search tree**.

3.3 Constructing Heaps

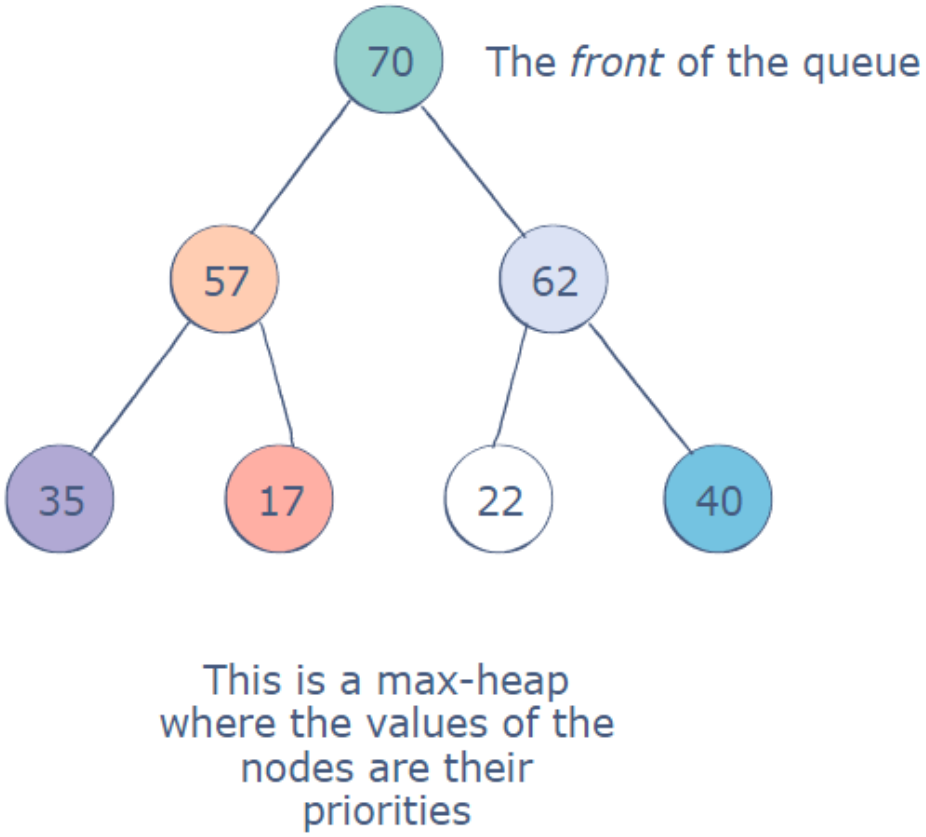
Heaps can be constructed incrementally, by inserting each new element into the left-most open spot in the array; namely $ (n + 1)st $ position of a previously n-element heap.

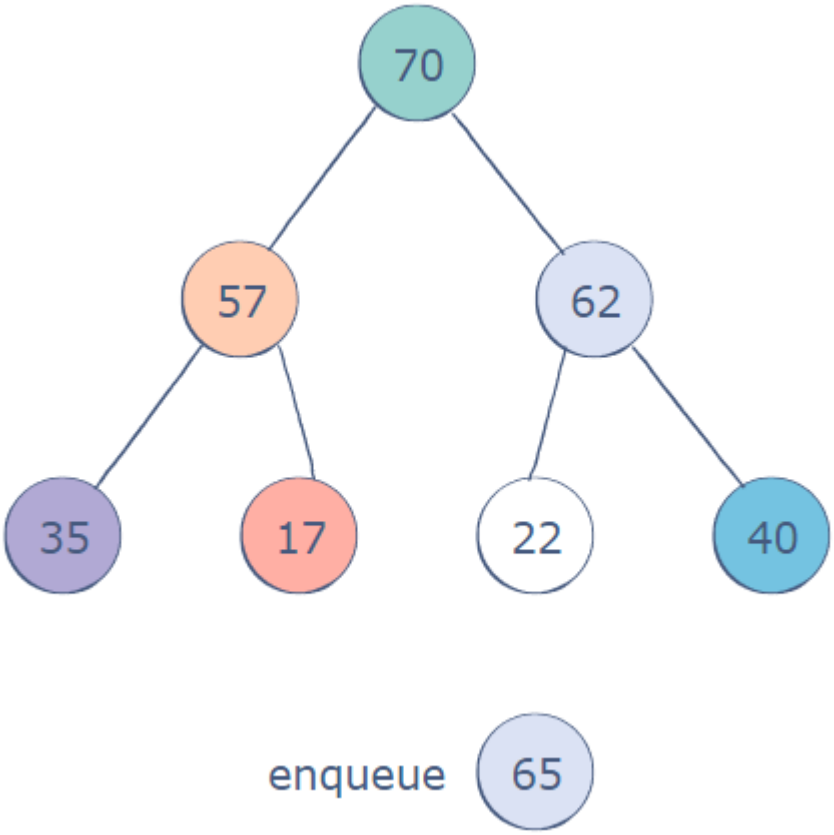
This ensures the desired balanced shape of the heap tree, but does not necessarily maintain the dominance ordering of the keys. The new key might be less than its parent in a min-heap, or greater than its parent in a max-heap

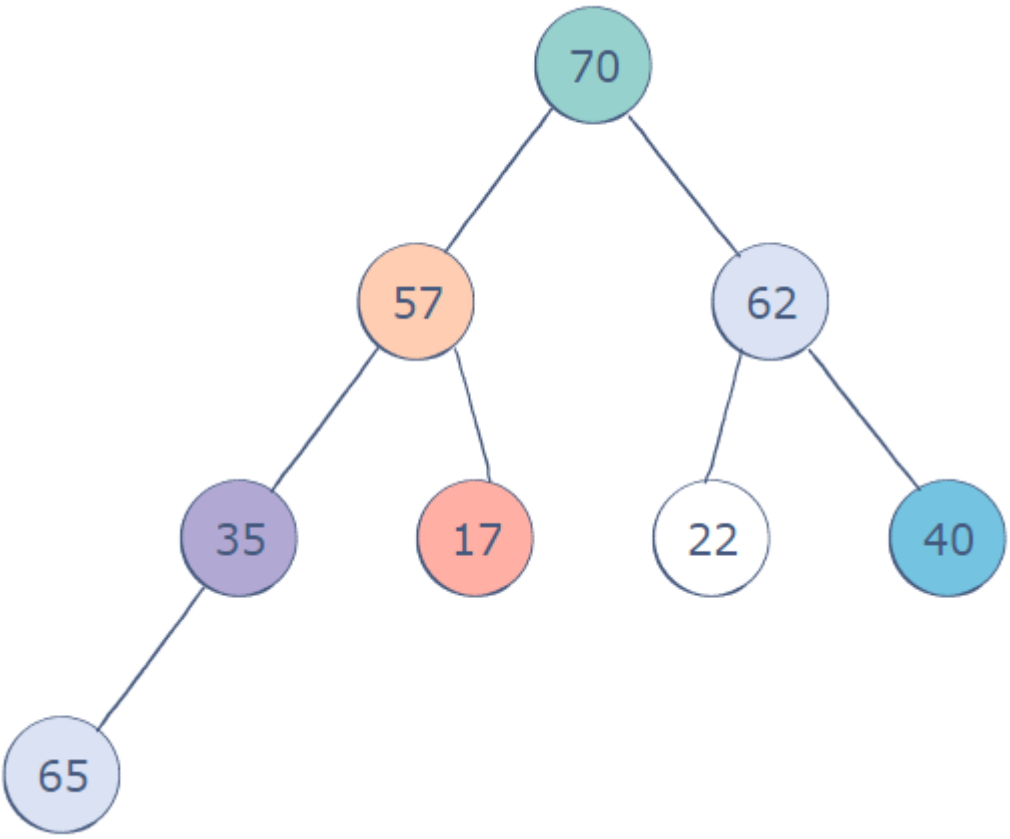
The solution is to swap any such dissatisfied element with its parent. The old parent is now happy, because it is properly dominated. The other child of the old parent is still happy, because it is now doiminated by an element more extreeme than its previous parent. The new element is now happier, but may still dominate its new parent

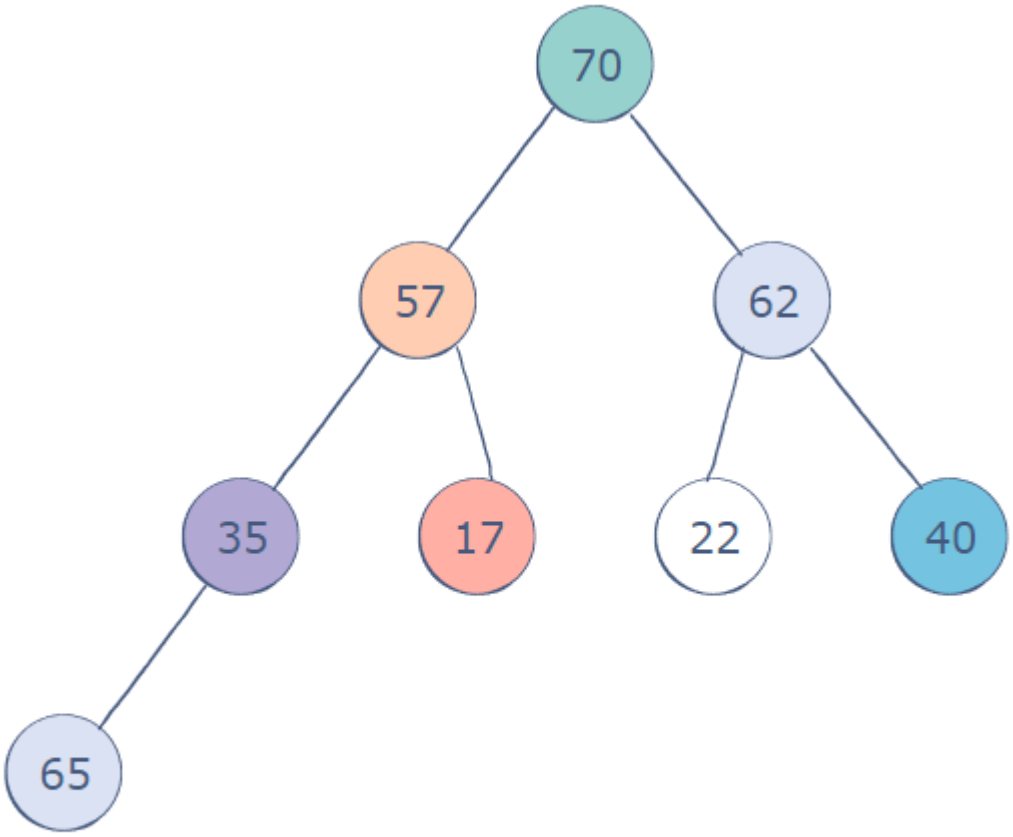
We now recur at a higher level, bubbling up the new key to its proper position in the hierarchy. Since we replace the root of the subtree by a larger one at each step, we pereserve the heap order.

3.3.1 Constructing Heap Steps

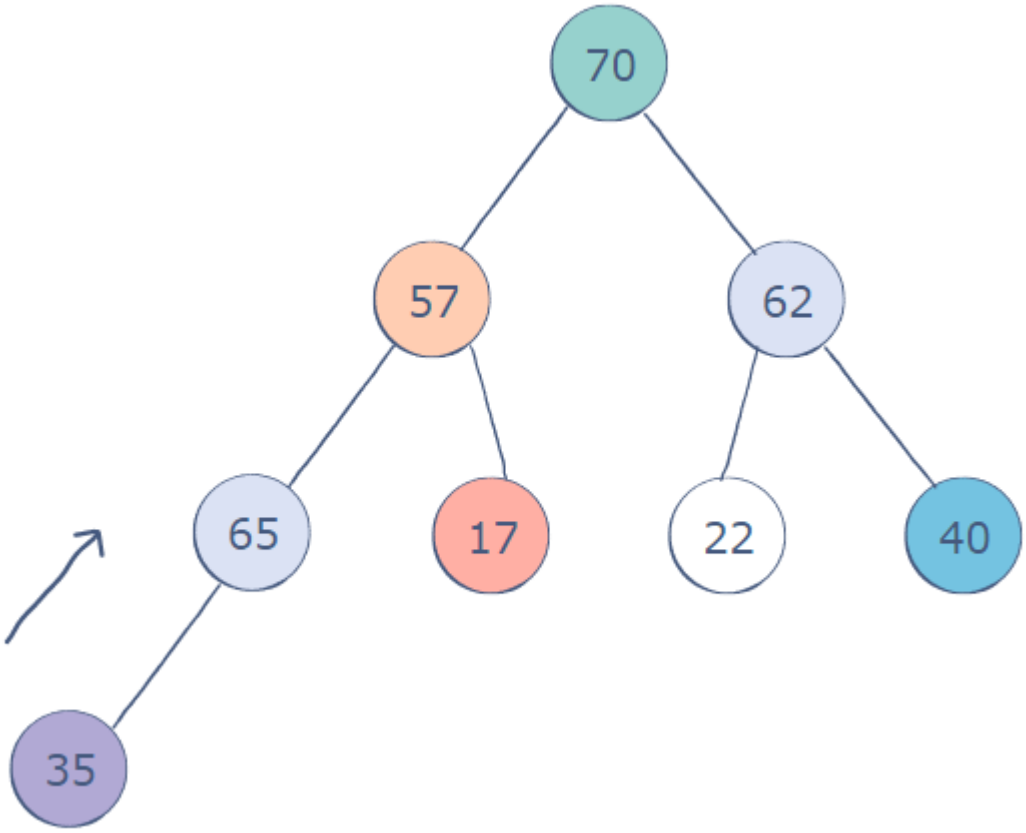


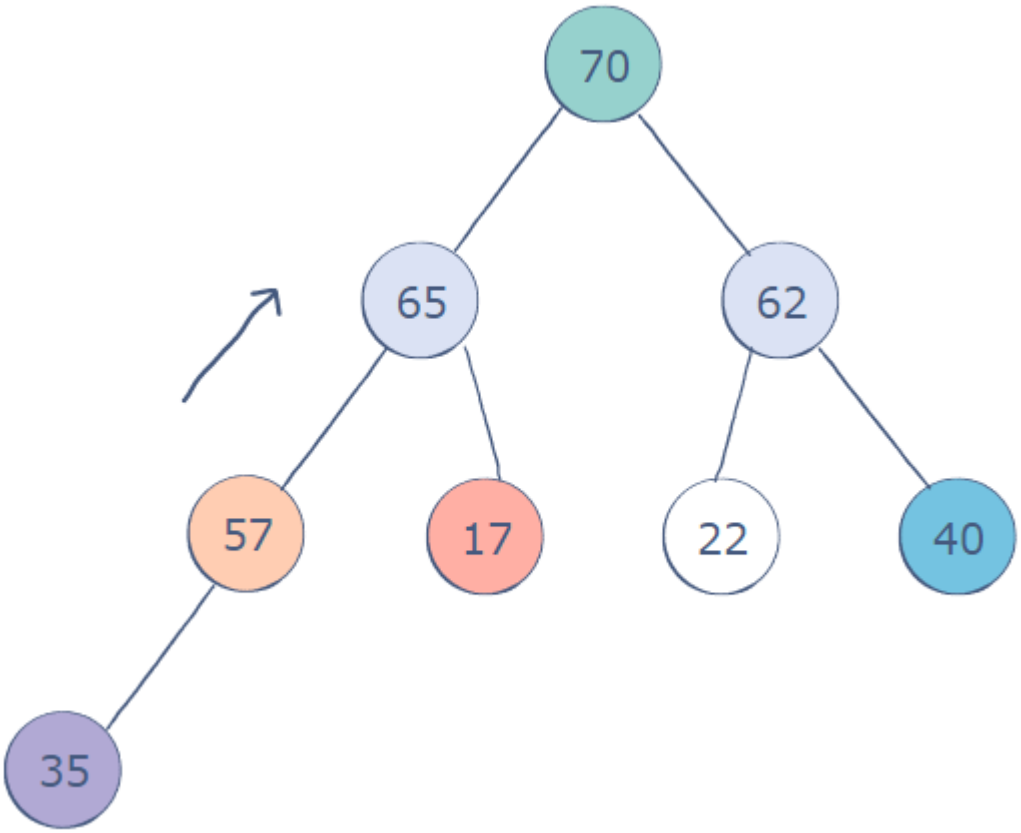

70
57
62
35
17
22
40
enqueue
65

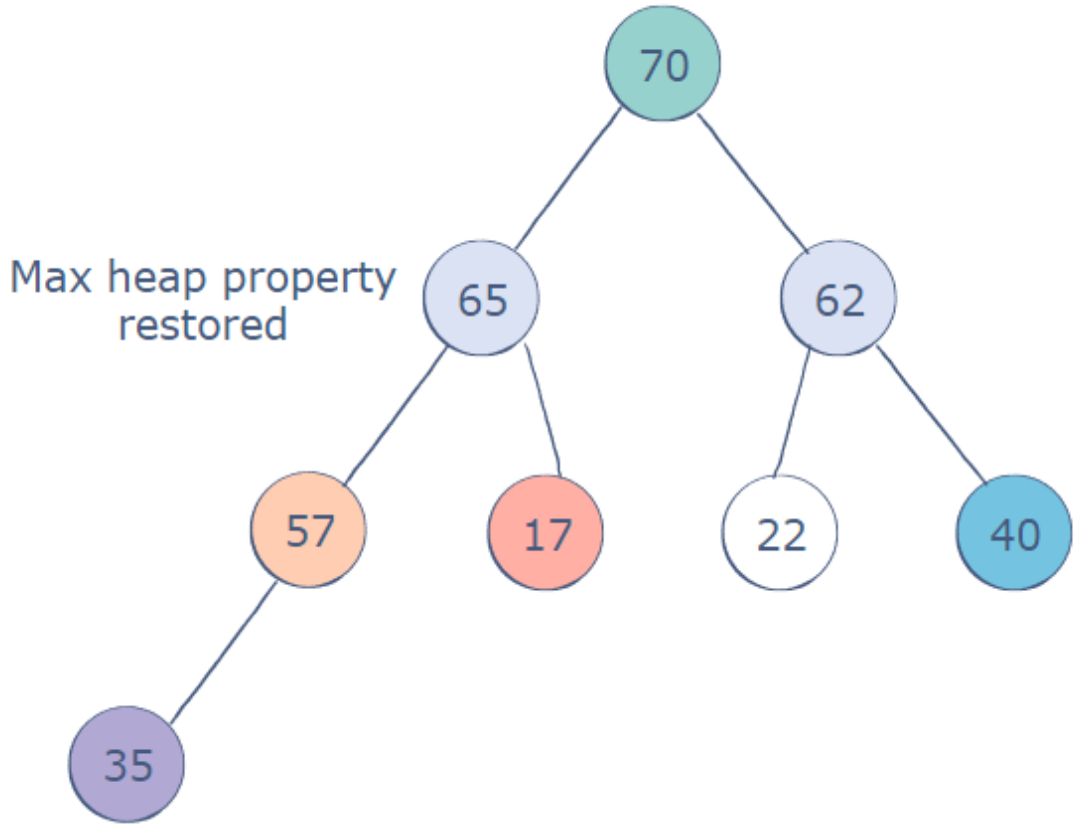

70
57
62
35
17
22
40
65



Swap with the parent node until the parent node has a higher value


70
57
62
65
17
22
40
35


70
65
62
57
17
22
40
35




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

bubble_up(priority_queue *q, int p)
{
    if (pq_parent(p) == -1) return; /* at root of heap, no parent */

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

The swap process takes constant time at each level. Since the height of an $ n $ element heap is $ [lg n] $, each insertion takes at most $ O (log n) $ time. Thus an initial heap of $ n $ elements can

be constructed in $ O (n log n) $ time though $ n $ usch isertions

```
pq_init(priority_queue *q)
{
    q->n = 0;
}

make_heap(priority_queue *q, item_type s[], int n)
{
    int i; /* counter */
    pq_init(q);
    for (i=0; i<n; i++)
        pq_insert(q, s[i]);
}
```

3.3.2 Extracting the Minimum

The remaining operations are identifying and deleting the dominant element.

Identification is easy since the top of the heap sits in the first position of the array. Removing the top element leaves a hole in the array. This can be filled by moving the element in the $ right-most $ leaf (sitting in the $ nth $ position of the array) into the first position. The shape of the tree has been restored but the key of the root may no longer satisfy the heap propert and the relevent swaps need to be made. The dissatisfied element *bubles down* the heap untill it dominates all its children, even perhaps by becoming a leaf node and ceasing to have any

The *peroclate-down* operation describes above is called **heapify** because it merges two heaps (the subtrees below the orginal root) with a new key

```
item_type extract_min(priority_queue *q)
{
    int min = -1; /* minimum value */

    if (q->n <= 0) printf("Warning: empty priority queue.\n");
    else {
        min = q->q[1];

        q->q[1] = q->q[ q->n ];
        q->n = q->n - 1;
        bubble_down(q,1);
    }

    return(min);
    }

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

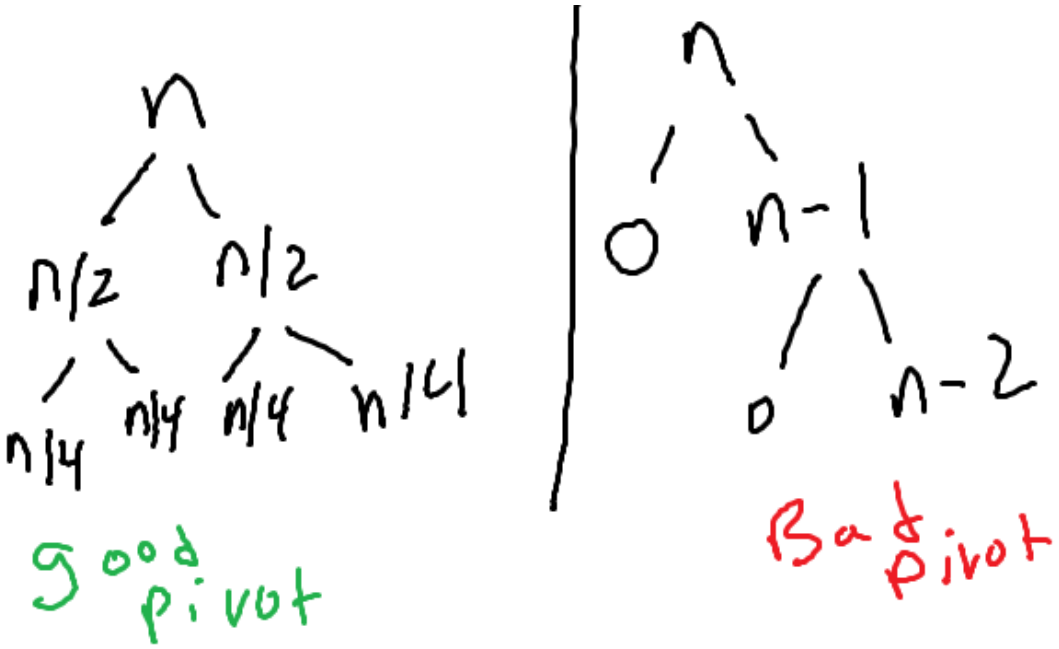
```
    c = pq_young_child(p);
    min_index = p;

    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);
    }
}
```

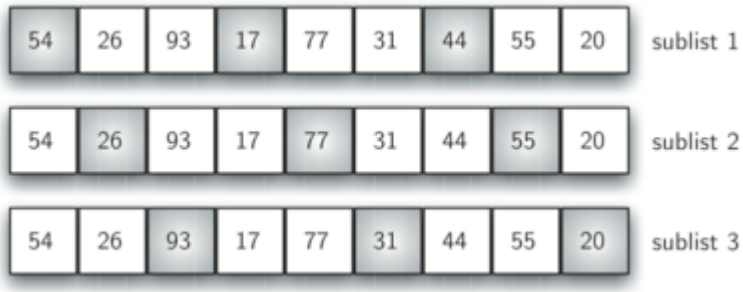
We will reach a leaf after $ [lg n] $ **bubble_down** steps, each constrant time. Thus troot deletion is completed in $ O (log n) $ time

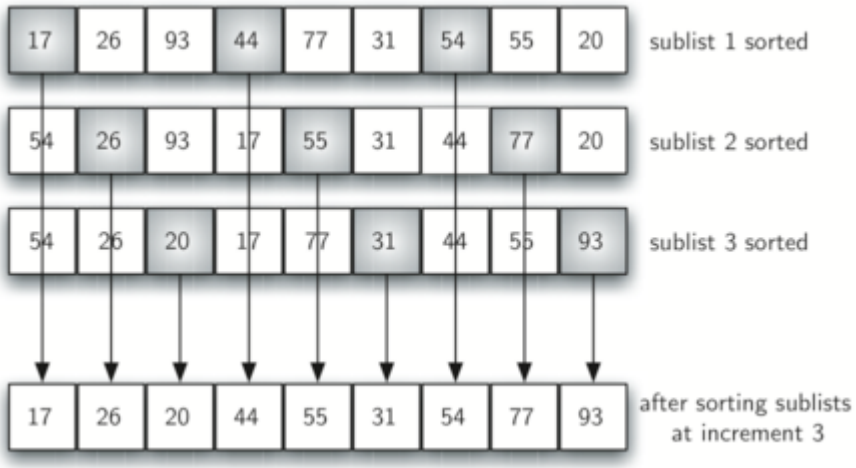
Exchanging the maximum element with the last element and calling **heapify** repeatedly gives an $ O (n log n) $ sorting algorithm, named **Heapsort**

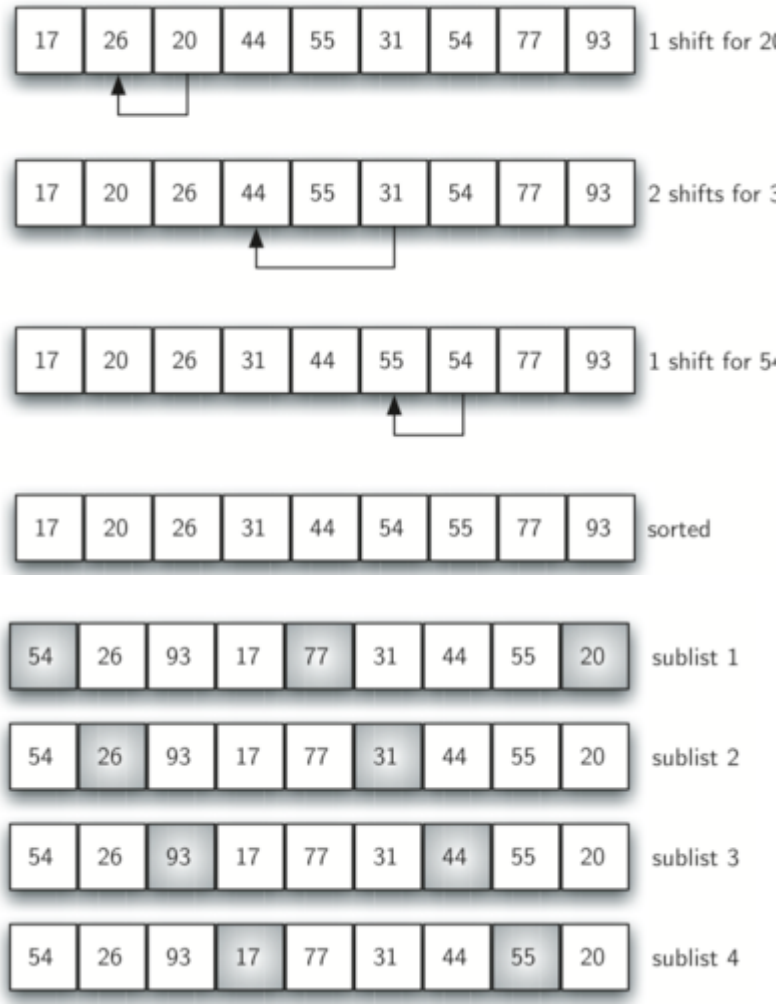
```
heapsort(item_type s[], int n)
{

int i; /* counters */
priority_queue q; /* heap for heapsort */

make_heap(&q,s,n);

for (i=0; i<n; i++)
    s[i] = extract_min(&q);
}
```

- Heapsort takes advantage of the heap datastructure which is an implementation of a priortiy queue. Since the top of the heap must be a min or max value, we can simply take the top value and begin moving it around. The purpose of heapsort is to efficiently sort an unsorted array. We do this via proxy tree. Simply take an array, transform it into a heap. This means that the largest value or smallest value is at the top, but not that the entire heap is sorted. So what we do is take the largest value and begin to move it down to the leaf node. Now the largest item in the heap is located at the last node, which is great. We know that it is in its sorted position, so it can be removed from the heap completely. But theres still another step, making sure that the new root node elements in the correct place! It highly unlikely that the item that we swapped into the root node position is in the right location, so we'll move down the root node position is in the right location; this bubbling down, is called heapify. The algorithm repeats self untill we arrive at just a single node.


n
n/2
n/2
n/4
n/4
n/4
n/4
Good pivot
n
0
n-1
0
n-2
Bad pivot


54 26 93 17 77 31 44 55 20
sublist 1
54 26 93 17 77 31 44 55 20
sublist 2
54 26 93 17 77 31 44 55 20
sublist 3


17 26 93 44 77 31 54 55 20
sublist 1 sorted
54 26 93 17 55 31 44 77 20
sublist 2 sorted
54 26 20 17 77 31 44 55 93
sublist 3 sorted
17 26 20 44 55 31 54 77 93
after sorting sublists at increment 3



Heapsorts worst-case is $ (n log n) $ which is the best you can hope for in a sorting algorithm

3.3.3 Fast Heap constuction

Heaps can be constructed faster than $ O (n log n) $ time by using **bubble_down** procedure

If we pack the $ n $ keys into for our heap into the first $ n $ elements of our priority-queue array. The shape of our heap will be correct but the dominance order will be messed up. To restore it, we have to consider the last $ nth $ position. It represents leaf of the tree and so dominates its nonexistant children. Same canse for the $ n/2 $ positions in the array, because all are leaves. If we continue to walk backwards, through the array we will finally encounter an internal node with children. The element may not dominate its children, but its children represent well-formed (if small) heaps

```
make_heap(priority_queue *q, item_type s[], int n)
{
int i; /* counter */

q->n = n;
for (i=0; i<n; i++) q->q[i+1] = s[i];
```

```
for (i=q->n; i>=1; i--) bubble_down(q,i);
}
```

Multiplying the number of calls to **bubble_down** $ (n) $ times an upper bound on the cost of each operation $ O(log n)) $ gives us a running time analysis of $ O (n log n)$. This would make it no faster than the incremental insertion algorithm described above, but note this is indeed an upper bound because insetion will take $ [lg n] $ steps

Bubble down takes time proportial to the height of the heaps it is merging. Most of these heps are extremely small. In a full tree of $ n $ nodes, there are $ n/2 $ nodes that are leaves, $ n/4 $ nodes that are height, $ n/8 $ nodes that are height $ 2 $ and so on.

In general, there are at most $ [n/2^{h+1}] $ nodes of hight $ h $, so the cost of builiding a heap is

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

The series quickely converges to linear because the puny contribution of the numerator $ h $ is crushed by the $ 2^h $

3.3.4 Where is the heap

Problem: Given an array based heap on $ n $ elements and a real number $ x $, effciently determine whether the $ kth $ smallest element in the heap is greater than or equal to $ x $. Your algorithm should be $ O (k) $ in the worst-case, independent of the size of the heap. You do not need to have to find the $ kth $ smallest elemen, but only determine its relationship to $ x $

Bad Solutions: - Call exact-min $ k $ times, and test wheter all of these are less than $ x $. This explicitly sorts the first $ k $ elements and so gives us more information than the desired answers, but it takes $ O (k log n) $ time to do so - The $ kth $ smallest element cannot be deeper than the $ kth $ level of the heap since the path from the root must go rhough elements of decreasing value. Thus we can look at all the elements on the first $ k $ levels of the heap and count how many of them ar less than $ x $, stopping when we either find $ k $ of them or run out of elements. This is correct, but takes $ O (min(n, 2^k)) $ time, sicne the top $ k $ elements have $ 2^k $ elemtns

An $ O(k) $ solution can look at only $ k $ elements smalelr than $ x $, plus at most $ O(k) $ elements greater than $ x $. COnsider the following procedure called at the root with $ i = 1 $ with $ count = k $

```
 int heap_compare(priority_queue *q, int i, int count, int x)
{
if ((count <= 0) || (i > q->n) return(count);

if (q->q[i] < x) {
    count = heap_compare(q, pq_young_child(i), count-1, x);
```

```
    count = heap_compare(q, pq_young_child(i)+1, count, x);
    }
    return(count);
}
```

The procedure searches the children of all nodes of weight smaller than $ x $ untill either $ (a) $ we have found $ k $ of them, when it return $ 0 $, or $ (b) $ they are exhausted, when it returns a value greater than zero; thus it will find enough small elements if they exist. The onyl nodes whos children we will look at are those $ < x $ and at most $ k $ of these in total. Each have at most visited two children, so we visit at most $ 3k $ nodes, for a total of $ O(k) $ time

3.3.5 Sorting By Incremental Insertion

Select an arbitrary element from the unsorted set, put it in the proper position in the sorted set

```
InsertionSort(A)
    A[0] = -ϖ
    for i = 2 to n do
        j = i
        while (A[j] < A[j - 1]) do
        swap(A[j],A[j - 1])
        j = j - 1
```

Althoug insertion sort takes $ O(n^2) $ in the worst case, it performs considerably better if the data is almost sorted, since few iterations of the inner loop suffice to shift it into proper position

Insertion sort is the simplest example of *incermental sort* technique.

Fastest sorting algorithms based on incremental insetion follow from more efficient data structures. Insertion into a balanced search tree takes $ O (log n) $ per operation or a total of $ O (n log n) $ time to construct the tree. An in-order traversal reads through the elements in sorted order to complete the jon in linear time

4 MergeSort: Sorting By Divide and Conquer

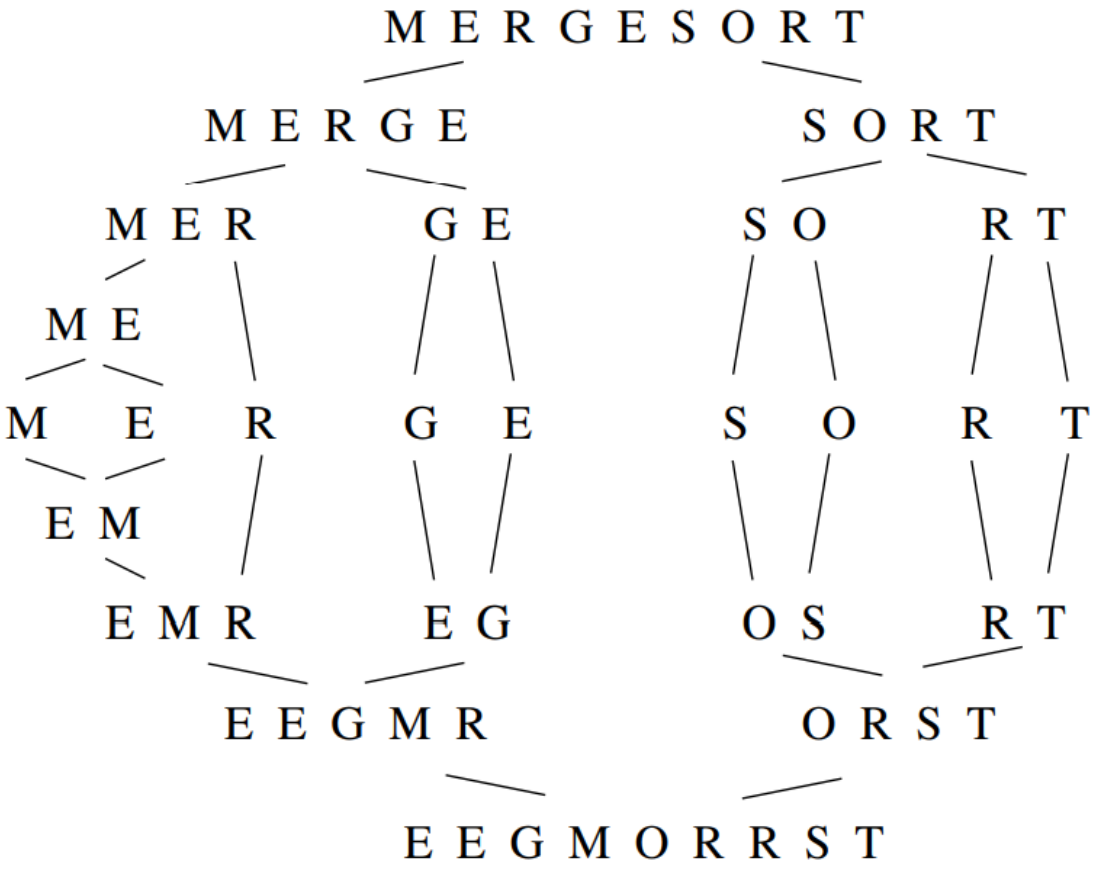


Figure 3: Animation of MergeSort in Action

A recursive approach to sorting involves partitioning the elements into two groups, sorting each of the smaller problems recursively and then interleaving the two sorted lists to totall order the elements

```
Mergesort(A[1,n])
    Merge( MergeSort(A[1, [n/2]), MergeSort(A[n/2 + 1,n]) )
```

The efficency of mergesort depends on how we combine the two sorted halves into a single sorted list. We could add them to a list and do heapsort but that destroys all the work spent sorting the individual component lists

So instead we *merge* the two lists. Repeating the merge operation untill both lists are empty merges two sorted lists into one (with a total of $ n $ elements betwen them) into one, using at most $ n - 1$ comparisons or $ O(n) $ total work.

If we assume for simplicitly that $ n $ is a power of $ 2 $, the $ kth $ level consists of all $ 2^k $ calls to **mergesort** processing subranges fo $ n/2^k $ elements

The work done on the $ (k = 0)$th level involves merging two sorted lists, each of size $ n/2$, for a total of at most $ n - 1$ comparisions. The work done on the $ (k = 1)$th level invovles merging two pairs of sorted lists, each of $ n/4 $, for a total of at most $ n - 2 $ comparisons. In general, the work done on the $ kth $ level invovled merging $ 2^k$ pairs sorted lists, each of size $ n/2^{k

+1} $, for a total of at most $ n - 2^k $ comparisons. *Linear work is done merging all the elements on each level*

the expensive work with the most comparsions is at the top level. the number of elements in a subproblem gets halved at rach leve. This the number of times we can halve $ n $ untill we get $ 1 $ is $ [lg_2 n] $. Since the recursion goes $ lg n $ level deep, a linear amount of work is done per level.

Merge sort takes $ O (n log n) $ time in the worst case

Mergesort is a good algorithm for sorting linked lists, because it does not rely on random access to elements as does heapsort or quicksort. The disadvantage is the need for an auxilliary bufffer when sorting arrays. To merge two sorted arrays, we need to use a third array to store the resuls

4.1 Implementation

MergeSort Pesudo-Code

```
mergesort(item_type s[], int low, int high)
{
    int i; /* counter */
    int middle; /* index of middle element */

    if (low < high) {
        middle = (low+high)/2;
        mergesort(s,low,middle);
        mergesort(s,middle+1,high);
        merge(s, low, middle, high);
        }
    }
```

MergeSort Implementation

```
merge(item_type s[], int low, int middle, int high)
{
    int i; /* counter */
    queue buffer1, buffer2; /* buffers to hold elements for merging */

    init_queue(&buffer1);
    init_queue(&buffer2);

    for (i=low; i<=middle; i++) enqueue(&buffer1,s[i]);
    for (i=middle+1; i<=high; i++) enqueue(&buffer2,s[i]);

    i = low;
    while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
    if (headq(&buffer1) <= headq(&buffer2))
        s[i++] = dequeue(&buffer1);
    else
        s[i++] = dequeue(&buffer2);
}
```

```
while (!empty_queue(&buffer1)) s[i++] = dequeue(&buffer1);
while (!empty_queue(&buffer2)) s[i++] = dequeue(&buffer2);
}
```

https://www.educative.io/edpresso/merge-sort-in-python

5 QuickSort: Sorting By Randomization

Suppose we select a random item $ p $ from the $ n $ items we seek to sort. Quicksort seprates the $ n - 1 $ other items into two piles: a low pile containing all the elements that appear before $ p $ in sorted order and a high pile containing all the elements that appear after $ p $ in sorted order. Low and high denote the array positions we place the respective piles, leaving a single slot between them for $ p $

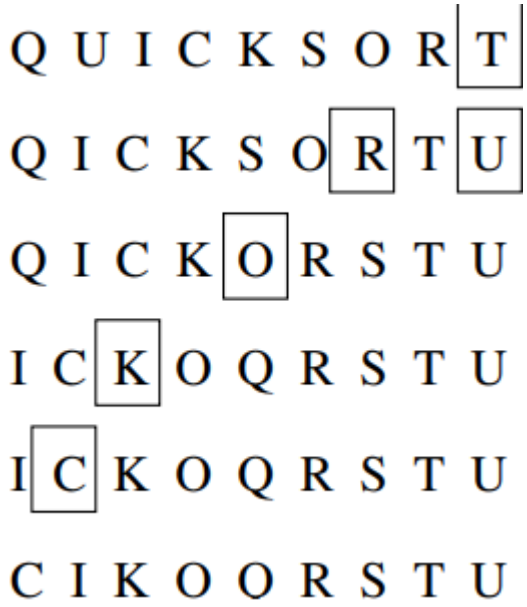


Figure 4: Animation of quicksort in action

The partitioning does two things. First, the pivot element $ p $ ends up in exact array position it will reside in the final sorted order. *Thus we can now sort the elements to the left and the right of the pibot independently!*. This gives us a recursive sorting algorithm, since we can use the partitioning approach to sort each subproblem

Quicksort Pesudo-Code

```
quicksort(item_type s[], int l, int h)
{
    int p; /* index of partition */

    if ((h-l)>0) {
        p = partition(s,l,h);
        quicksort(s,l,p-1);
        quicksort(s,p+1,h);
```

```
    }
}
```

We can partition the array in one linear scan for a particular pivot element by maintaining three sections of the array: less than the pivot (to the left of `firsthight`), greater than or equal to the pivot (between `firsthight` and `i`) and, unexplored (to the right of `i`) as implemented below:

```
int partition(item_type s[], int l, int h)
{
    int i; /* counter */
    int p; /* pivot element index */
    int firsthigh; /* divider position for pivot element */

    p = h;
    firsthigh = l;
    for (i=l; i<h; i++)
        if (s[i] < s[p]) {
            swap(&s[i],&s[firsthigh]);
            firsthigh ++;
        }
    swap(&s[p],&s[firsthigh]);

    return(firsthigh);
}
```

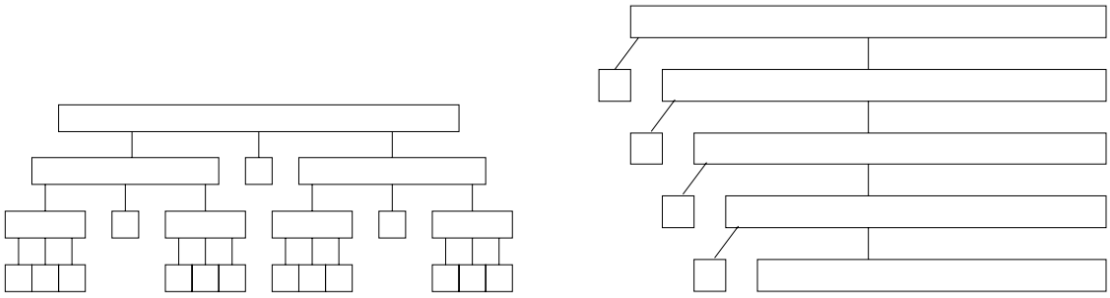


Figure 5: The best-case (l) and worst-case $(r) $ recursion trees for quicksort . This is a recursion tree representation

Partitioning steps consists of at most $ n $ swaps, it takes linear time in the number of keys

As with mergesort, quicksort builds a recursion tree of nested subranges of the $ n $-element array. Instead of merging, your partitioning

Also, mergesort and quicksort both run in $ O(n*h) $ time, where $ h $ is the height of the recursion tree.

The difficult is that the height of the tree depends upon where the pivot element ends up in each partion. If we are luckey and *happen* to repeatedly pick the median element as our pivot, the subproblems always half the size of the previous level. The height represents the number of times we can halve $ n $ untill we get to $ 1 $, which is $ [lg_2 n] $.

If we get unlucky, we keep having to split the array as unequally as possible. Implies the pivot

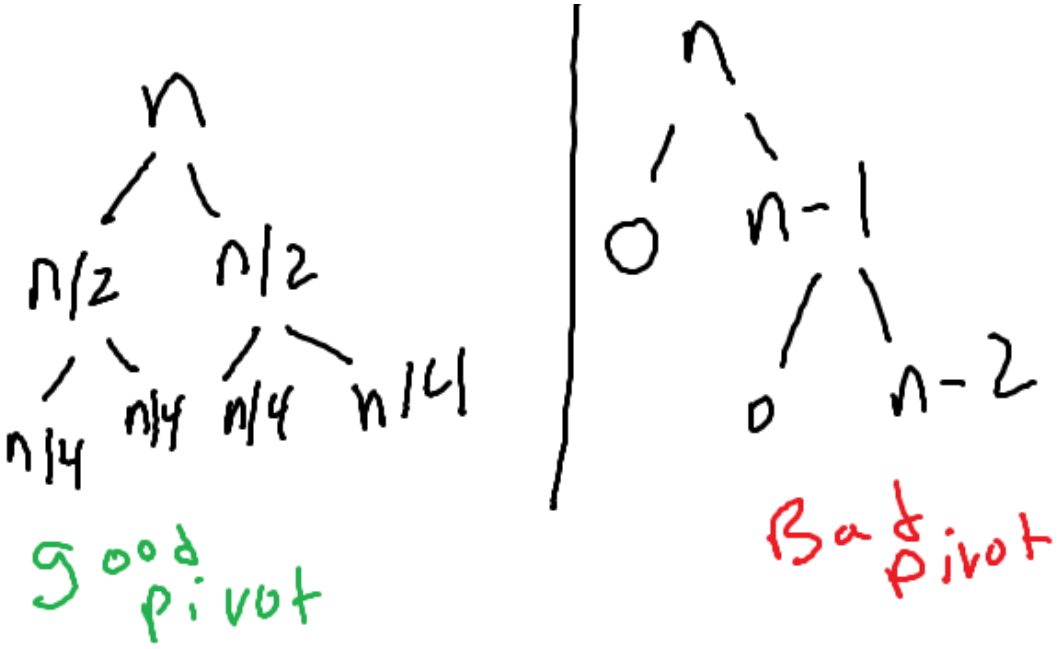
element is always the biggest or smallest element in the sub-array

Aftet pivot settles into position, we are left with one subproblem of size $ n - 1 $. We spent linear work and reuced the problem by one measly element

It takes a tree height $ n - 1 $ to chop our array down to one element per level, for a worst case of $ O (n^2) $

Thus, quicksorts worsr case is worse than heapsort or mergesort, but average case is better

- Important to note that, the worst case for quicksort is $ n^2 $ because we ended up picking a terrible pivot. Imagine if you picked the smallest number of the largest number. In that case, you doing $ n - 1 $ comparisions. Then you keep picking the terrible number. On the other hand, imagine you picked a pivoit that is the medium, or the middle number. Then your splitting all the numbers less then the pivot on one half and all the numbers greater then the pivot on the other half. Now it basically becomes a problem of cutting in half each time. Halved? Well it Log time. in this case, the complexity is $ n log n $



5.1 The expected case for quicksort

Figure 6: Half the time, the pivot is close to the median element

How likely is it that a randomly selected pivot is a good one? The best possible selection for the pivot would be the median key, because exactly half of the elements would end up left and half the elements right, of the pivot.

We only have a $ 1/n $ probability of selecting the median as pivot, which is small

What if a key is good enough if the pivot if it lies in the center helf of the sorted space of keys, meaning those ranked from $ n/4 $ to $ 3n/4 $ in the space of all keys to be sorted

Such pivots are good enough since half the elements lie closer to the middle than one of the two ends. Thus on each selection, we will pick a good enough pivot with a probability of $ 1/2 $

The worst possible enough pivot leaves the bigger half of the space partition with $ 3n/4 $ items. The height of a quicksort partition tree constructed repeatedly from the worst-possible good noug pivot is 40% taller

The deepest path throug the tree passes though partions of sizes $ n, (3/4)n, (3/4)^2n, ...$ down to 1. How amny time can we multiply $ n $ by $ 3/4 $ untill it gets down to $ 1 $? Since the expected number of good splits and bad splits is the same, the bad splits only double the height of the tree, so

$$(3/4)^h_g n = 1 \; n = (4/3)^h_g$$

$$h_g = log_{4/3} n$$

$$h = 2h_g = 2log_{4/3} n = O(logn)$$

since quicksort does $ O (n) $ work partitioning on each level, the average time is $ Θ (n log n) $

The odds against $ O (n^2) $ are vanishingly small

5.2 Randomized Algorithms

The thing about quicksort runtime being $ Θ (n log n)$ is that it requires a random selection as a pivot. But if you were to sort the array first then select the pivot, you would end with a nightmare $ O (n^2) $

But if we were assured randomness, we could say "Randomized quicksort runs in $ Θ(n log n) $ time on any input with high probability"

Randomization is a powerful tool to improve algorithms with bad worst-case but good average-case complexity

Randomizing Algorithms: - Random Sampling - Want to get an idea of the median value of $ n $ things but dont haave either the time or space to look at them all? Select a small random sample of the input and study those for the results should be representative. - Random Hashing - We have claimed that hashing can be used to implement dictionary operations in $ O (1) $ "expected-time." However for any hash fucntion there is a given worst-case set of keys that all get hashed to the same bucket. But now suppose we randomly selected our hash function from a large family of good ones as the first step of the algorithm. We get the same type of improved guarantee that we did with randomized quicksort - Randomizes Search - Randomization can also be used to drive search techniques such as simulated annealing

5.3 Nuts and Bolts

Problem: You are given a collection of n bolts of different widths, and $ n $ corresponding nuts. You can test whether a given nut and bolt fit together, from which you learn whether the nut is too large, too small, or an exact match for the bolt. The differences in size between pairs of nuts or

bolts are too small to see by eye, so you cannot compare the sizes of two nuts or two bolts directly. You are to match each bolt to each nut. Give an $O(n^2)$ algorithm to solve the nuts and bolts problem. Then give a randomized $O(n log n) $ expected time algorithm for the same problem

Randomized quicksort is perfect for this. We ask if we can parition the nuts into bolts around a randomly selected bolt $ b$. Once we find the matching nuts to $ b $ we can use it to partition the bolts. In $ 2n- 2 $ comparisions, we partition the nuts and bolts. Then we do randomized quicksort.

5.4 Is Quick Sort Really Quick?

Mergesort, heapsort and quicksort should all outperform insertion sort or selection sort on large enough instances

A good quicksort algorithm is 2-3 times faster than mergesort or heapsort

6 Distribution Sort: Sorting via Bucketing

If we sorted names according to the first letter of the last name, we will create 26 piles or buckets of names. Then we sort by character $ (A, B, …)$ and then merge in the end; we are still sorting each bucket individually before we combine them. The names will be sorted when their is a single name in the bucket. The resulting algorithm is called **bucketsort** or **distribution sort**

Bucketing is very effective when we know that the distribution of data will be roughly uniform.

It underlies hash tables, $ kd$-trees and other practical data structures.

Down side is that the performace is terrible when the data distribution is not what we expected.

7 Lower Bounds for Sorting

You cannot. $ Ω(n log n) $ always exists. A sorting algorithm must behave differently during execution on each of the distinct $ n ! $ permutations of $ n $ keys. The outcome of each pairwise comparsion governs the run-time behavior of any comparison-based sorting algorithm

We can think of the set of all possible executions of such an algorithm as a tree with $ n! $ leaves. The min height tree corresponds to the fastest possible algorithm that happens

$$lg(n!) = (nlogn)$$

8 Binary Search and Related Algortihms

Binary search is a fast algorithm for searching in a sorted array of keys $ S $. To searchh for key $ q $, we compare $ q $ to the middle key $ S[n/2] $. If $ q $ appears before $ S[n/2] $, it must reside in the top half of $ S $; if not, it must reside in the bottom half of $ S $. By repeating the process recursively on the correct half, we locate the key in a total $ [lg n]$ comparisons, a big win over the $ n/2 $ comparisons except when using sequential search

```
int binary_search(item_type s[], item_type key, int low, int high)
{
```

```
    int middle; /* index of middle element */

    if (low > high) return (-1); /* key not found */

    middle = (low+high)/2;

    if (s[middle] == key) return(middle);

    if (s[middle] > key)
        return( binary_search(s,key,low,middle-1) );
    else
        return(binary_search(s,key,middle+1,high) );
}
```

Using binary srarch, we can find a word in a $50,000$ to $200,000$ word dictionary in $ 20 $ iterations

8.1 Counting Occurances

A variant of binary search

Suppose we wanted to count the number of times a given key $ k $ occurs in a given sorted array. Because sorting groups all the copies of $ k $ into a contigous block, the problem reduces to finding the right block and then measures its size

The binary search routine presented above enables us to find the index of an element of the correct block $ x $ in $ O (lg n) $ time. The natural way to identify the boundaries of the block is to sequentially test elements to the left of $ x $ untill we find the first one that differs from the search key, and then repeat this search to the right of $ x $. The differences between the indices of the left and right boundaries (+1) give the count of the number of occurances of $ k $.

The algorithm runs in $ O (lg n + s)$, when $ s $ is the number of occurances of the key.

A faster algortihm results by modifying binary search to search for the boundary of the block containing $ k $, instead of $ k $ itself

Suppose we delete the equality test

```
if (s[middle == key) return (middle);
```

from the implementation above and return the index **low** instrad of -1 on each unsuccessful search. All searches will now be unsuccessful, since there is no equality test. The search will proceede to the right half whenever the key is compared to an identical array element, eventually terminating at the right boundary. Repeating the search after reversing the direction of the binary comparison will lead us to the left boundary.

Each search takes $ O (lg n) $ time, so we cn count the occurances in log time regardless of the size of the block

8.2 One-Sided Binary Search

Suppose we have an array $ A $ consisting of a run of $ 0$'s, followed by an unbounded run of $ 1$'s and would like to identify the exact point of transition between them. Binary search on the

array would provide the transition point in $ [lg n] $ tests, if we had a bounded $ n $ on the number of elements in the array.

In the absence of a bound, we can test repeatedly at larger intervals ($ A[1], A[2], A[4], A[8], A[16], ...$) untill we find a first non-zero value. We then have a window containing the target and can proceed with binary search

The *one-sided binary search* finds the transition point $ p $ using at most $ 2[lg p] $ comparisons, regardless of how large the array actually is. One-sided binary search is most useful whenever we are looking for a key that lies close to our current position

8.3 Square and Other Roots

Square root of $ n $ is the number $ r $ such that $ r^2 = n $. First observe that the square root $ n >= 1 $ must be at least 1 at most $ n $. Let $ l = 1 $ and $ r = n $. Consider the midpoint of this interval $ m = (l + r)/2 $

How does $ m^2 $ compare to $ n $? If $ n >= m^2 $, then the square root must be greater than $ m $, so the algorithm reprats with $ l = m $.

If $n < m^2 $, then the square toot must be less than $ m $, so the algortihm repeats with $ r = m $. Either way we have halved the intervals using only one comparision.

Therefore, after $ lg n $ rounds, we will have identified the square root within $ +- 1 $

This method is called the Bisection Method.

8.4 Summary of Binary Search

Binary search and its variants are the quintessential divide-and-conquer algorithms

9 Divide and Conquer

Divide and conquer splits the problem in halves, solves each half and then stiches the pieces back together to form a solution

When ever merging takes less time than solving the two subproblems, we get an effcient algorithm

Mergesort and binary search are perfect examples of a divide-and-conquer algortihm

Our ability to analyze divide-and-conquer algortihms rests on our strenghts to solve the asymptotic of recurrences relations governing the cost of such recursive algortihms

9.1 Recurrence Relations

A recurrance relation is an equation that is define din terms of itself.

The fibonacci numbers are described by the natural functions are easily expressed by recurrences

$$F_n = F_{n-1} + F_{n-2}$$

Any polynomial can be represented by a recurrence, such as the linear function

$$a_n = a_{n-1} + 1, a_1 = 1 \rightarrow a_n = n$$

Any exponential can be represented by a reccurrence

$$a_n = 2a_{n-1}, a_1 = 1 \rightarrow a_n = 2^{n-1}$$

Finally, lots of weird functions that cannot be described easily with conventional notation can be represented by recurrence:

$$a_n = na_{n-1}, a_a = 1 \rightarrow a_n = n!$$

Recurence relations provide a way to analyze recursive structure, such as algorithms

9.2 Divide-And-Conquer Recurrences

Divide-and-conquer algortihms tend to break a given problem into some number of smalelr pieces (say $ a$), each of which is size $ n/b $.

Further, they spend $ f(n) $ time to combine these subproblem solutions into a complete result.

Let $ T(n) $ denote the worst case time the algortihm takes to solve a problem of size $ n $.

Then $ T(n) $ is given by the following recurrence relations

$$T(n) = aT(n/b) + f(n)$$

Examples: - Sorting: - The running time behavior of mergesort is governed by the currence $ T(n) = 2T(n/2) + O(n)$, since the algorithm divides the data into equal-sized halves and then spends linear time merging the halves after they are sorted. Infact, this recurrence evalutes to $ T(n) = O(n lg n) $ just as we got by our previous analysis - Binary Search - The running time behavior of binary search is governed by the recurrence $ T(n) = T(n/2) + O(1)$, since at each step we spend constant time to reduce the problem to an instant half its size, in fact, this recurrence evaluates to $ T(n) = O(lg n) $, just as we got by the previous analysis - Fast Heap Construction - The **bubble_down** method of heap construction built by $ n $-element heap by constructing two $ n/2 $ element heaps and then merging them with the root in logarithmic time. This argument reduces to the reccurence relation $ T(n) = 2T(n/2) + O (lg n)$. In fact, this recurrence evaluates to $ T(n) = O(n)$, just as we got by our previous analysis - Matrix Multiplication - The standard matrix multiplication algorithm for two $ n * n $ matrix takes $ O(n^3) $, becuase w compute the dot product of $ n $ terms for each of the $ n^2 $ elements in the product matrix

9.3 Solving Divide-and-Conquer Recurrences

Divide-and-conqur recurrences of the form $ T(n) = aT(n/b) + f(n) $ are generally easy to solve because they fall into three cases

9.3.1 Case 1

if

$$f(n) = O(n^{log_b a - dx})$$

for some constant $ dx > 0 $, then

$$T(n) = (n^l og_b a)$$

Case 1 hods for heap construction and matrix multiplication

9.3.2 Case 2

if

$$f(n) = (n^{log_b a})$$

then

$$T(n) = (n^{log_b a} lgn)$$

Case 2 holds for mergesort and binary search

9.3.3 Case 3

if

$$f(n) = (n^{log_b a - dx})$$

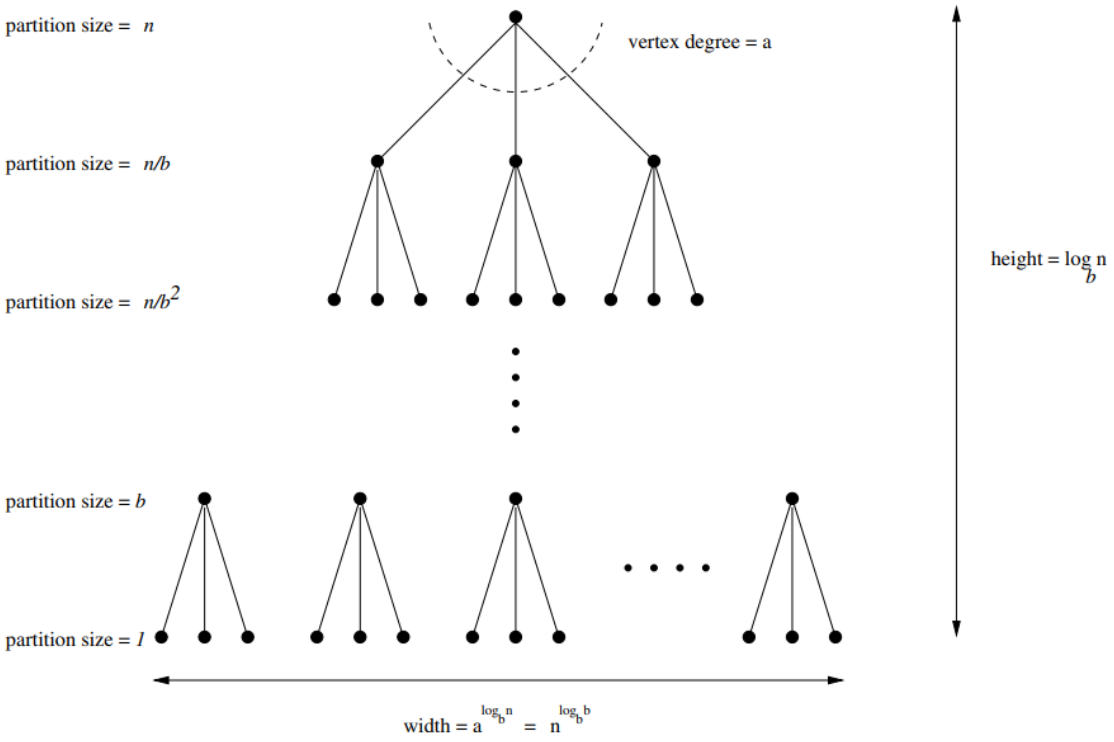
for some constant $ dx > 0 $

and if $ af(n/b) <= cf(n) $, some $ c < 1 $ then

$$T(n) = (f(n)$$

Case 3 generally arises for clumsier algorithms, where the cost of combining the subproblems dominates everything

the three cases are refered to as *master theorms*



Figrue 7: The recursion tree resulting from decomposing each problem of size $ n $ problmes of size $ n/b $

The figure above shows the recursion tree associated with typical $ T(n) = aT(n/b) + f(n) $ divide and conquer algorithms

Each problem of size $ n $ is decomposed into a problem of size $ n/b$

Each subproblem of $ k $ takes $ O(f(k)) $ time to deal with internally, between partitioning and merging.

The total time for the algorithm is the sum of these internal costs, plus the overhead of building the recursion tree

The height of this tree is $ h = log_bn $ and the overhead of building the recursion tree

The height of this tree is $ h = log_bn $ and the number of leaf nodes $ a^h = a^{log_bn}$, which happens to simplify to $ n^{log_ba} $

The three different cases of the mastr theorm correspond to three different costs which might be dominant as a function of $ a, b $ and $ f(n) $

cases: - 1. Too Many leaves: if the number of a leaf node outweights the sum of the internal evaluation cost, the total running time is $ O(n^{log_ba}) $ - 2. Equal work per level: As we move down the tree, each problem gets smaller but there are more to sovle. If the sum of the internal evaluation costs at each level are qual, the total running time is cost per level $ n^{log_ba} $ times the number of levels $ log_bn $ for a total running time of $ O(n^{log_ba}lgn $ - 3. Too expensive a root: if the internal evaluation costs grow rapidly enough with $ n $, the cost of the root evaluation may dominate, if so, the total running time is $ O(f(n)) $

10 ShellSort

A more effective version of Insert sort. Also called diminishing increment sort

It improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort.

The unique way that these sublists are chosen is the key to the shell sort. Instead of breaking the list into sublists of contigous items, the shell sort uses an increment $ i $, sometimes called the gap, to create a sublist by choosing all items that ate $ i $ items apart

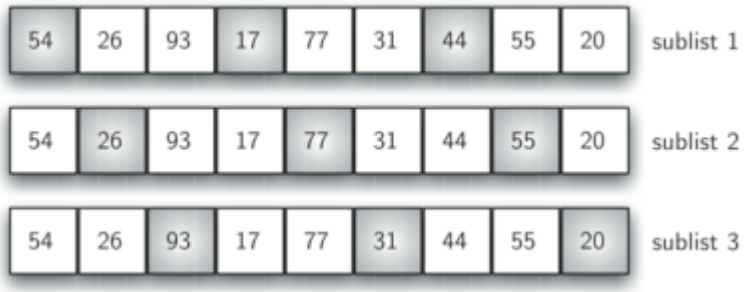


Figure 8: A shell sort with increments of three

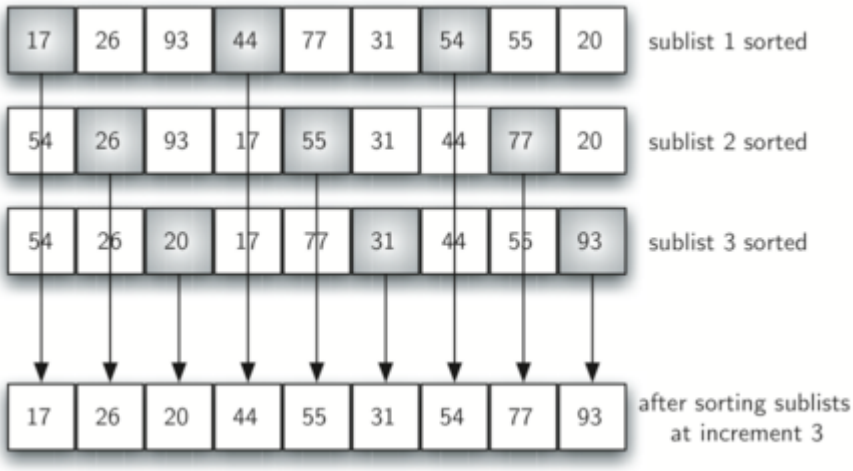


Figure 9: A shell sort after sorting each sublists

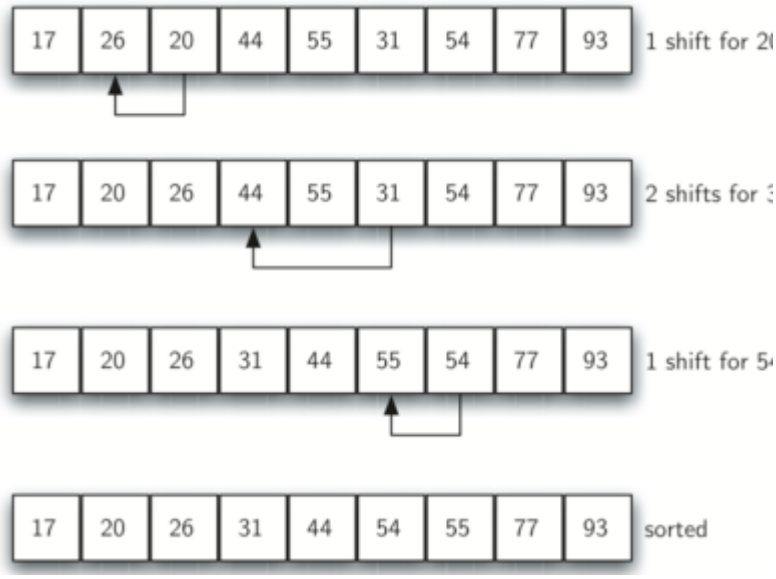


Figure 10: Shellshort: A final insertion sort with incremental of 1

Figure 11: Initial Subslits for a shell Sort

[1]:
```
def shellSort(alist):
    sublistcount = len(alist)//2
    while sublistcount > 0:

      for startposition in range(sublistcount):
        gapInsertionSort(alist,startposition,sublistcount)

      print("After increments of size",sublistcount,
                                   "The list is",alist)

      sublistcount = sublistcount // 2

def gapInsertionSort(alist,start,gap):
    for i in range(start+gap,len(alist),gap):
```

```
        currentvalue = alist[i]
        position = i

        while position>=gap and alist[position-gap]>currentvalue:
            alist[position]=alist[position-gap]
            position = position-gap

        alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
shellSort(alist)
print(alist)
```

```
After increments of size 4 The list is [20, 26, 44, 17, 54, 31, 93, 55, 77]
After increments of size 2 The list is [20, 17, 44, 26, 54, 31, 77, 55, 93]
After increments of size 1 The list is [17, 20, 26, 31, 44, 54, 55, 77, 93]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Shell short tends to fall between $ O(n) $ and $ O (n^2) $

11 Redix Sort

Work in progress

QuickSort in Python https://www.educative.io/edpresso/how-to-implement-quicksort-in-python

https://www.youtube.com/watch?v=uXBnyYuwPe8

https://levelup.gitconnected.com/a-sort-of-all-sorting-algorithms-506cbc76d47