Recursion versus Iteration

- Space (in-place or out-of-place?)

Practical Issues

- Worst case versus average case

Theoretical Issues

- Different sorting algorithms illustrate different ideas
  - A complete analysis can often be done
  - The application itself is easy to understand
  - Pedagogically useful
  - Sorting is a big deal

Reasons for Studying

Sort

5
- Locality of reference?
- Memory to memory? Disk to disk?
- Number of moves
- Sort keys, not data
- External sort versus internal sort
{ 
  { 
    
    e[0] = [a] 
    e[1] = [a] 
    } 

  
  if (a[i] < [a][j]) 
  comparisons++; 
    } 

  for (i = 0; i < count; ++i) 
  { 
    
    for (j = i + 1; j < count; ++j) 
    { 
      
        if (count == 1) 
        { 
          temp[j] = temp[i]; 
        } 

        if (temp[j] < temp[i]) 
        { 
          temp[i] = temp[j]; 
          comparisons++; 
        } 

        if (count == count) 
        { 
          temp[i] = temp[j]; 
        } 

        if (temp[j] == temp[i]) 
        { 
          temp[i] = temp[j]; 
        } 

        temp[j] = temp[i]; 
    } 

  } 

  { 
    
    e[0] = [a] 
    e[1] = [a] 
    } 

  } 

}
{ }
{ }
break;
else
  { 
    temp = [i]+1
    a[i] = a[i+1]
    temp = a[i]
  }
  if(a[i] < [i+1]"comparisons++;");")
for(i = length-1; i >= 1; i--)
  
for(i = length = 2; i < length ? hommany : length++; 
  if(hommany == count) hommany--;
  if(hommany, temp);
  
(void insert(int a[], long count, long hommany, long count, long *comparisons)
  longer by one with each iteration // to keep track of the total number of comparisons // sort the array a[], *count of length, count // sort only the first hommany sort steps // note that this inserts the last element into the sorted list // to create a sorted initial portion of the list that gets // insert into the list that already contains the initial portion // sort insertion sort

5.2 Insertion Sort
```c
{
    
    if (compare(a[i+1], evaluate))
        {
            /*
             * if 
             * 
             * a[compare] = [a[i+1]
             * 
             * comparisons++;
             */

            for (length = 2; length <= hommany; length++)
                if (hommany == count)
                    hommany--;
                else
                    insert[evaluate] = length;
            }
        
        /*
         * sort the array a["," count of length, compare
         * sort the array a["," count of length, compare
         * perform only the first, hommany sort steps
         * keep track of the total number of comparisons
         * this differs from the previous version only in that
         * the last down and create the vacancy into which we
         * we don't store every step; we only push
         * with eventually store the element to be inserted
         */
```
induction. Note that this requires that every pair be out of order.

For every $n$, there exists a permutation of $\binom{n}{2}$ inversions. This can be proved either directly or by

**Proposition** of inversion is not at all the same as that of a transposition.

For those of you who know about permutation groups, note that this derives

\[ (4,3,2,1). \]

This inversion

For example, the list

but $w(i) < w(j)$.

**Definition** 5.1. Given a permutation $\pi$ on $n$ integer keys, an inversion is a pair $(i, j)$ such that $i > j$.

The nature of sorting algorithms is to find the order that any list of elements is some permutation of its sorted

With some sorting algorithms we can prove lower bounds on the complexity of algorithms. We will describe

5.4 Lower Bounds
many inversions, and the theorem is proved.

Average case: Consider permutations in pairs of a permutation and its reverse. For each of the $n!/2$ pairs of elements, each pair is an inversion in exactly one of the two permutations, so the $n! - 1$ inversions are shared between the two permutations. On average, then, a random permutation has half that value.

Worst case: For every $u$, there exists a permutation with $n! - 1$ inversions, so if only one is removed with each comparison, then worst case running time must require $n! + 1$ comparisons.

Worst case running time of at least $n! - 1$ comparisons.

Theorem 5.2. A sorting algorithm that sorts by comparison of keys and that requires at most one
(\text{proof of the corollary}) Insertion sort has the effect of comparing adjacent elements in a list and exchanging them. However, the effect is no different from what would happen if the element to be inserted were in fact inserted into the vacancy at every step, and then it would be true that adjacent elements instead create a vacancy. Instead, create a vacancy. But when they are out of order, (\text{proof of the corollary}) Insertion sort has the effect of comparing adjacent elements in a list and exchanging them and average case running time of at least $\frac{1}{4}n(n-1)$ comparisons.

Corollary 5.3. Insertion sort has lower order worst case running time of at least $\frac{1}{2}n(n-1)$ comparisons.
when we run out of one list, copy the other list to merged-list

```c
while (ptr-merged-list != null)
    // copy the smaller (from list j) to the merged list
    if (list-j[ptr-j-list-1] > list-i[ptr-i-list-2])
        merged-list[ptr-merged-list++] = list-j[ptr-j-list-1];
    else
        merged-list[ptr-merged-list++] = list-i[ptr-i-list-2];
```

The algorithm:

- Merge two lists using three pointers
- This requires additional space (i.e., can't be done in place)

If we have two sorted lists, each of size \(n/2\), we can merge them with no more than \(n - 1\) comparisons. But
order of the pairs in question.

But reversing either pair above does nothing to change the sorted order of the final array except for the

cannot correctly sort two arrays in which these elements are reversed in magnitude.
If for some reason the algorithm is written so as not to compare $q_i$ with $a_i^f$ for some $i$, then the algorithm

cannot correctly sort two arrays in which these elements are reversed in magnitude.
If for some reason the algorithm is written so as not to compare $q_i$ with $a_i^f$ for some $i$, then the algorithm

we must compare $q_i$ with $a_i^f$ for $1 \leq i \leq \frac{n}{2}$, and that then we must do $n - 1$ comparisons. The claim is that we must compare $q_i$ with $a_i^f$ for $1 \leq i \leq \frac{n}{2}$.

\[
\begin{align*}
\tilde{v}/u &> \tilde{v}/u_d > \ldots > 1+i/q_i > 1+i/q > 1/q_i > 1/q > \ldots > \tilde{v} > \tilde{v}/u_d > 1/q_i > 1/q > 1/n
\end{align*}
\]

Proof. Assume arrays $A$ and $B$. We claim that $[q_i] = [u_i]$ and $[q_i] = [u_i] = [u_i]^f$.

Theorem 5.4. Any algorithm that merges by comparison of keys two sorted arrays of $n/2$ elements

The worst case: we never run out of either list because the two lists exactly interleave.
3. and then merge the two lists

2. sort those two lists by merge sort

1. while $n \geq 2$, split the list into two lists of $n/2$ items

The merge sort algorithm: To sort $n$ items:

5.6 Merge sort
\[
(u \Theta \Theta) = \\
(I - u) - u(u \Theta l) + (I)l \cdot u = \\
...
\]

\[
\zeta - u \varepsilon + (8/u)l \cdot 8 = \\
(I - 4/u)4 + \varepsilon - u \zeta + (8/u)l \cdot 8 = \\
\zeta - I - u \zeta + (4/u)l \cdot 4 = \\
(I - 1/4)2 + I - u + (4/u)l \cdot 4 = \\
I - u + (\zeta/u)l \cdot 2 = (u)l
\]

Proof: Clearly 

Theorem 5.5: Mergesort has worst case running time of \(Θ(n \log n)\).
3. and it doesn't cache well.

2. the constant is worse than other constants.

1. it's a fixed cost algorithm—there's no benefit to having the list sorted to start with.

On the other hand:

3. and it doesn't require extra space.

2. average case is $O(n)$

1. worst case is $O(n)$

Heap sort in theory is an outstanding sort:

5.7 Heap sort
This last property (4) is called the **max-heap property**.

children exist.

4. the key value at any node is greater than or equal to the key value of either of its children, if any

3. if a leaf at depth h is missing, then all leaves to its right are also missing;

2. all leaves are at depth h or depth h – 1;

1. T is complete at least through depth h – 1;

**Definition 5.6.** A binary tree T is a max-heap if and only if
A heap as a binary tree.
Example.

Contrast this with the linear time of a linked list, for
of the heap data structure as a heap in Le u time. Insertion into a list of elements in Le u time, removal of the next element in constant time, and reconstruction
Another use for a heap is in a priority queue. If the only need is for the highest priority element in a

| * | * | * | * | * | 3 | 9 | 11 | 17 | 23 | 38 | 16 | 25 | 37 | 52 | 72 |

Or, if the binary tree is not a complete tree, something like

| 37 | 52 | 16 | 25 | 19 | 38 | 9 | 11 | 17 | 23 | 3 | 8 | 22 | 33 | 72 |

IMPORTANT: A heap as a linear array.

Example 5.7.2
7. Further, this would be guaranteed free cost, running time.

6. Then we can put the elements into sorted order in another log n steps.

5. So if we can create a heap in the first place in log n steps.

4. Resulting in a heap with one element less than before.

3. And then recreate the heap in log n steps by pushing the used-to-be-last element into its proper place.

2. We can exchange the root with the “last” element.

1. The largest element is the root of the tree.

If we have a heap, then

5.8 Heapsort metaphysics
Heap to be used as a priority queue.

- Max-Heapify, Heap-Insert, Heap-Extract-Max, Heap-Increase-Key, and Max-Maximum allow a heap to be sorted in place in $O(n \log n)$ time using a heap.
- Heappop sorts $n$ items in place in $O(n \log n)$ time using a heap.
- Max-Heapify maintains a max-heap in $O(n \log n)$ time for each call.
- Build-Max-Heap creates a max-heap from an array in linear time.

We will use a heap and several functions that manipulate it:
{  
    {  
        max-heapify(a[], TARGET)  
        exchange a[i] and a[TARGET]  
        }  
    if(TARGET != i) {  
        if a child value is the TARGET, then exchange and return  
        {  
            TARGET = RR;  
        }  
        if(RR >= heap-size(a)) {  
            if(a[RR] < a[TARGET]) {  
                TARGET = RR;  
            } else  
                {  
                    TARGET = III;  
                }  
        } }  
    }  
    void max-heapify(a[], i)  
    {  
        push a[i] into its proper place to restore the full heap  
        // Assume that subtrees below node i are heaps  
    }

5.8.1 \textsc{Max-Heapify}
Following:

Looked at differently, in terms of number of nodes, the maximum imbalance in left and right subtrees is the worst case here is 2^x nodes and height y instead of height y - 1 if all were balanced.

* * * * * * *

6 16 25 19 38

12 37

72

This runs in worst-case time equal to the max depth of the tree.
Either way, max-heapify runs in \( \mathcal{O}(\log n) \) time.

This makes for \( u = 3 \cdot 2^h - 1 \) total nodes, of which \( 2^{h-1} \) are in the left subtree.

Thus height \( h \) we have \( 2^h - 1 \) nodes in the right subtree and \( 2^{h-1} - 1 \) in the left subtree.

* * * * 11 1 23 9

\[
\begin{array}{cccccc}
& & 38 & & & \\
& 32 & & & & \\
16 & & & & & \\
& 25 & & & & \\
& 19 & & & & \\
& & 72 & & & \\
\end{array}
\]
Termination: At the end of the iteration, we have $i = 0$. We have maintained the loop invariant throughout.

Iteration: We have created a heap with root at the $i$-th node.

This condition is satisfied. Max-heapify preserves the heap-property of children, so when we are finished with the children of node $i$ the children are both roots of heaps. This means that when we call max-heapify, all subtrees are a heap, and thus every such node is the root of a trivial heap of one element.

Maintenance: The children of node $i$ are numbered higher than $i$, since we are iterating down. Therefore, every node with a larger subtree is a heap, and thus every such node is the root of a trivial heap of one element.

Initialization: At the start of the first iteration, we have $i = \lfloor \log(\frac{n}{2}) \rfloor$. Every node with a larger subtree for subtrees of one element is a root of a max-heap.

Loop Invariant: At the start of the $i$-th iteration of the for loop, every node of the for loop, every node for subtrees larger than $i$ is a root of a max-heap.

```c
{
  {
    max-heapify(a, i);
  }
}
for (i = \floor{\log_{2}(\text{size}(a))}; i >= 1; i = i/2)
  heap-size(a) = \text{size}(a);
}

void build-max-heap(a[])
  build a heap of an array a

5.8.2 Build-Max-Heap
```
one can't push every element all the way to a leaf node.

The naive analysis assumes the worst case for every element, and thus does not take into consideration that

\[
(u)O = (u)O \geq \left( \frac{u^2}{\eta} \sum_{u \in \mathbb{N}} u \right) O
\]

with \( x = 1 \) where we get that the sum is bounded above by \( 2 \) and thus the cost is

\[
\frac{\varepsilon(x - 1)}{x} = x^x \sum_{0=\eta} \infty
\]

Using the generating function (A.8)

\[
\left( \frac{u^2}{\eta} \sum_{u \in \mathbb{N}} u \right) O = (u)O \left[ \sum_{1+\eta^2/u} \right] \sum_{[u \in \mathbb{N}]}
\]

heap the cost is

Heaps has height \( 2 \) and at most \( [u \in \mathbb{N}] \). Whenever we build the

dependent at height \( \eta \) elements at the root. However, a more careful analysis shows that in fact the running time is linear and not \( O \eta \). A heap of \( O \eta \) max-heapify procedure \( u \) times.

The build-max-heap procedure requires the time \( O(u \log u) \) (worst case, \( u \leq n \)). 

The iteration, and hence node 1, and all other nodes are the root of a heap.
time, and this cost dominates the linear time cost of build-max-heap.

\[ \text{Time: The algorithm takes } O(\log n) \text{ time, since it calls max-heapify } n - 1 \text{ times at cost } \frac{1}{2} \text{ each.} \]

```c
void heappsort(a[], t)
{
    heappsize(a);
    for (i = |a|; i > 2; i--)
        build-max-heap(a);

    for (i = |a|; i > 1; i--)
        exchange a[i] and a[1];
        repeat // required to put the new a(1) in its proper place,
    // rebuild the heap that we just put at the end,
    // then decrease the heap’s length by one (so as not to touch the
    // first element and last elements in the array,
    // so exchange the first and last elements in the array,
    // if we have a max heap, then the max is the first element,
    // build a max heap of the array
    // algorithm:
    // sort in items in array a
    //
    // 5.8.3 Heapsort}
```
Recompute distances, and then cluster again…

Data clustering algorithms often want to cluster data nodes based on the nearest pairwise distance.

A similar situation exists in scheduling events in an operating system.

Items scheduled for the next soonest execution time be executed at some future time. The simulation process itself waits to pull off the event list only those events scheduled for the next soonest execution time. The simulation process itself waits to pull off the event list only those events scheduled for the next soonest execution time.

Consider simulation (or the real time) of events. New events are created dynamically and scheduled to be executed at some future time. The simulation process itself waits to pull off the event list only those events scheduled for the next soonest execution time.

Key value queues require a priority queue, a data structure for maintaining a set of elements, each with an associated key value, so that they can be accessed in the sorted order of their key value.
Clearly constant time

```
{   return(a[1]);
}
```

Heaps:

- **Decrease-key** to decrease the priority (scheduled execution time) of the data item with the given key.
- **Increase-key** to raise the priority (scheduled execution time) of the data item with the given key.
- **Extract-min** to pull the smallest item in the list off the list.
- **Extract-max** to pull the largest item in the list off the list.
- **Maximum** to return the item in the list with the largest key value.
- **Maximum** to return the item in the list with the smallest key value.
- **Insert** a new data item into an existing list.
- **Delete** an item from a list
- **Search** an item in a list
- **Check** if an item is in a list
- **Size** of the list
- **Delete** all items from the list
Clearly \( O(n) \) time on an array of length \( n \)

```c
{
return (maxvalue);

max-heapify(a, i)
{
    heap-size(a) = heap-size(a) - 1;
    if (heap-size(a) < i)
        return;
    maxvalue = a[i];
    k = i;
    while (k < heap-size(a) / 2)
    {
        if (heap-max(a) > heap-max(a))
            swap(a[k], a[k + 1]);
        k = 2 * k;
    }
}

// a priority queue structure as a priority queue
// and then rebuild the heap to maintain the data
// decrease the length by 1
// move the last value to the first location
// extract the largest key value in the set
```
Note the importance of random access into the array, and compare this with a linked list.

Clearly $O(\log n)$ time on an array of length $n$

```c
{
    
    i = parent(i);
    exchange a[i] and parent(a[i]);
}

while (i < 1 && a[parent(i)] > a[i])
    a[i] = key;

{new key isn't an increase

if (key > a[i])
    void heap-increase-key(a, i, key);

maintain the priority queue // increase the value of a specific element, and
Heap-increase-key
```
Clearly $O(n \log n)$ time on an array of length $n$.

```c
void max-heap-insert(a, key)
    heap-size(a) = heap-size(a) + 1;
    heap-insert(a, key);
    max-heap-maintains(a) = heap-maintains(a);
```

5.9.4 Max-heap-insert
Assume the pivot element is the first element in the array. If it isn’t, then exchange the pivot element with

As for the rearrangement (P):

average case to choosing the midpoint element as our pivot element.

We can’t, of course, find an order. But if we’re lucky and careful, we might be able to come close in the

Then we could sort the entire array in time $O(n \log n)$.

Let’s assume that with each recursion, we can arrange the data as in (P) in time proportional to $n$.

The recursion depth $u$ items

In reality, involving the order at every stage to be able to choose the midpoint element every time.

Recursively, invoking the order at every stage to be able to choose the midpoint element every time.

Then call this procedure are less than $\frac{n}{2}$ and all items subscripted larger than $\frac{n}{2}$ are larger than $\frac{u}{2}$. Then call this procedure

Now, let’s assume that we rearrange the array of $u$ items so that all items subscripted less than $\frac{u}{2}$ be $\frac{u}{2}$.

Assume that we could (invoke or hide here) pick out the $\frac{u}{2}$-th element $a_{\frac{u}{2}}$ which we can re-index to

Let’s assume that we have $n$ items to sort.

Q.10 QuickSort
Assuming the order, then, we could guarantee \( \leq n \) running time.

and a second half larger than the pivot element.

**Bottom Time:** In time \( n \) we can cause the array to be split into a first half smaller than the pivot element.

At the end, the vacancy will be in the middle of the array, and we put our pivot element there.

- Put that in the vacancy, creating a first-half vacancy.
- Now work forward from position 2 until we find an element that is larger than the second half of the array.
- in the vacancy in position one, thus creating a vacancy in the second half of the array.
- Work from the end toward the front until we find the first element smaller than the pivot. Put that element
- the first element; pull the pivot element aside to create a vacancy in position one.
So we have a running time in the worst case. 

The k-th step.

In this case, we will do n recursion steps and have to compare the pivot element against n - k elements in splitting the array size in half each time. We would only be shortening the list by one element every time.

If our pivot element in fact was the smallest element, then we would not be doing a divide and conquer.

Without the oracle, how bad could this be?
Symmetric,
because the initial conditions happen to all the formula and because the $V(i)$ and $V(0)$ terms are

$$(*) \quad 1 \leq u \quad V \sum_{\ell=0}^{u} V + (\ell) \sum_{l=1}^{u} V + 1 - u = (u)V$$

This can be collapsed into

$$\leq u \quad V \sum_{\ell=0}^{u} V + (\ell) \sum_{l=1}^{u} V + 1 - u = (u)V$$

So the average case running time is

$$V(0) = 0.$$ 

We have $V(0) = 0$.

We do $n - 1$ comparisons in the initial split.

Our choice of pivot element is random, so it is the $\ell$th element for random $\ell$ and splits the array of length $n$.

Proof: Let $A(n)$ be the average case number of comparisons for sorting $n$ elements.

Theorem 5.7: Quick sort has average case running time for large $n$, of approximately $2 \log_2 n$. 

5.10.2 Average Case
What if we make a wild leap of faith that average isn't asymptotically worse than perfect?

The average running time down to \( n \) log \( n \).

We have already seen that if we had an oracle and could choose exactly the right pivot point, we could get
I. Choose \( c = \sqrt{3} - \varepsilon \).

This gives us an upper bound of \( \ln \log n \). To get the lower bound:

If we choose \( c \gg 2 \), then clearly the proposition is proved.

\[
u \left( \frac{c}{c} - 1 \right) + u \log u \cdot c \geq \]

\[
\left( \frac{uc}{1} - 1 \right) - u \left( \frac{c}{c} - 1 \right) + u \log u \cdot c =
\]

\[
\left( \frac{uc}{1} + \frac{c}{u} - u \log u \right) c + 1 - u =
\]

\[
\left( \frac{uc}{x} - \frac{c}{x \log x} \right) u + 1 - u =
\]

\[
x p \cdot x \log x \cdot c \int_{u}^{1} \frac{u}{c} + 1 - u \geq
\]

\[
\int_{1-u}^{1} u - u \geq
\]

\[
\log \sum_{1}^{1} u - u \geq
\]

\[
\forall u \geq 1 \quad \sum_{1}^{1} u - u = (u) \forall
\]

**Proof.** Induct. True for \( n = 1 \). Assume true \( \forall i \geq 1 \). From \( u \geq i \geq 1 \), we have \( \forall \log u \leq (u) \forall \), we have \( \forall 1 \leq i \cdot \)
\[
\left( \frac{\frac{\partial}{\partial t} \phi}{t} + \frac{\partial}{\partial x} \phi - (I - u) \frac{\partial}{\partial t} \phi (I - u) \right) \frac{u}{c^2} + I - u = \]

\[
\left( \frac{\frac{\partial}{\partial t} \phi}{t} + \frac{\partial}{\partial x} \phi - \frac{\partial}{\partial t} \phi (I - u) \right) \frac{u}{c^2} + I - u = \]

\[
\left[ \frac{\frac{\partial}{\partial t} \phi}{t} - \frac{\partial}{\partial x} \phi \right] \frac{u}{c^2} + I - u = \]

\[
\int_{-u}^{1-u} \frac{u}{c^2} + I - u \leq \]

\[
\int_{1-u}^{1} \frac{u}{c^2} + I - u = \]

\[
\int_{1-u}^{1} \frac{u}{c^2} + I - u < \]

\[
(u) \int_{1-u}^{1} \frac{u}{c^2} + I - u = (u) V \]

**Specifically:** Let \( c = 2 - \varepsilon \). Then

the function. The integral of the function bounded from above by rectangles intersecting at the right point.

an increasing function is bounded below by rectangles whose height is the left point of intersection with the integral.

3. Shift the boundaries on the integral to make it go the other way. This inequality says that the integral of

2. Rearrange the proposition with \( V(u) < c \) and use
term on the right-hand side therefore looks like $n$ times a constant for large $n$, and the second term looks like

Now, as $u$ goes to infinity, $(I - u) \frac{u (u / I - 1)}{I - 1} = 0$, and therefore our natural logarithm goes to $-1$. The first

$$(I - u) \ln (I - u) - u \ln (u / I - 1) \ln u =$$

$$(I - u) \ln (I - u) - \frac{u}{I - u} \ln u =$$

$$(I - u) \ln (I - u) - u \ln u - (I - u) \ln \varepsilon (I - u)$$

So let's look at

and the inequality holds.

term is actually of smaller order of magnitude than $u$, which will show that the $\varepsilon (I - u) \ln (I - u) - u \ln u - (I - u) \ln \varepsilon (I - u)$

Now, what we will show is that the

$$u \ln u - (I - u) \ln \varepsilon (I - u) \ln u < (I - (I - u) \ln \varepsilon (I - u) \ln u \ln u - \varepsilon (I - u) \ln \varepsilon (I - u) \ln \varepsilon (I - u)$$

When we substitute $\varepsilon - z$ and then rearrange terms, we get

$$u \ln u \ln u < \left( \frac{z}{I} + \frac{z}{\varepsilon (I - u)} - (I - u) \ln \varepsilon (I - u) \right) $$

We want this last value to be larger than $\ln u$. That is, we want to have

$$u \ln u < \left( \frac{z}{I} + \frac{z}{\varepsilon (I - u)} - (I - u) \ln \varepsilon (I - u) \right) u + I - u$$
\[
Z - uZ + (I - u)\mathcal{V}Z = \\
(2 - u)(I - u) - (I - u)\mathcal{V}Z + (I - u)\mathcal{V}u = (I - u)\mathcal{V}(I - u) - (u)\mathcal{V}u
\]

Subtract the last of these from the previous, and we get
\[
I \geq u \quad (\mathcal{V})\sum_{\mathcal{V}, \mathcal{V}}^I \mathcal{V}Z + (Z - u)(I - u) = (I - u)\mathcal{V}(I - u)
\]

and thus
\[
I \geq u \quad (\mathcal{V})\sum_{\mathcal{V}, \mathcal{V}}^I \mathcal{V}u = (u)\mathcal{V}u
\]

so that
\[
I \geq u \quad (\mathcal{V})\sum_{\mathcal{V}, \mathcal{V}}^I \frac{I - u}{\mathcal{V}} + \mathcal{V} - u = (I - u)\mathcal{V}
\]

and thus
\[
I \geq u \quad (\mathcal{V})\sum_{\mathcal{V}, \mathcal{V}}^I \frac{u}{\mathcal{V}} + 1 - u = (u)\mathcal{V}
\]

We have

**Proof** (Second proof by X. Yliao)

\[\square\]

So we're done.

That big, and the necessary expression above in is true for \( u \) sufficiently large.
\[
\left(\frac{\varepsilon - u}{\bar{z} - u}\right) \bar{z} + (\bar{v} - u) V \left(\frac{\varepsilon - u}{\bar{z} - u}\right) = (\varepsilon - u) V
\]

\[
\left(\frac{\bar{z} - u}{\varepsilon - u}\right) \bar{z} + (\varepsilon - u) V \left(\frac{\bar{z} - u}{\bar{z} - u}\right) = (\bar{z} - u) V
\]

\[
\left(\frac{I - u}{\bar{z} - u}\right) \bar{z} + (\bar{z} - u) V \left(\frac{I - u}{\bar{z} - u}\right) = (I - u) V
\]

\[
\left(\frac{u}{I - u}\right) \bar{z} + (I - u) V \left(\frac{u}{I + u}\right) = (u) V
\]

Thus

\[
\bar{z} - u\bar{z} + (I - u) V (I + u) = (u) V u
\]

so that
The magnitude of \(2^n\) log \(u\)

so if we integrate rather than sum, we'll get a log \(u\) factor with a multiplier of \(1\) and a genuine order of \(u\)

\[
\frac{1 + x}{1} + \frac{\zeta + x}{\zeta}
\]

sum of

The first term is order \(u\). The second term is \(1 - \frac{1}{u}\) and thus bounded by a constant (say \(Z\)). The sum is a

\[
\left(\frac{(I + ?)(\zeta + ?)}{\zeta - u}\right) \sum_{\zeta = u}^{1 = ?} (1 + u) \zeta + \left(\frac{u}{1 - u}\right) \zeta + (1) V \left(\frac{\zeta}{1 + u}\right) =
\]

\[
\left(\frac{(I + y - u)(\zeta + y - u)}{y - u}\right) \sum_{\zeta = y - u}^{\zeta = y} (1 + u) \zeta + \left(\frac{u}{1 - u}\right) \zeta + (1) V \left(\frac{\zeta}{1 + u}\right) =
\]

\[
\left(\frac{3 - u}{v - u}\right) \left(\frac{\zeta - u}{1 + u}\right) \zeta + \left(\frac{3 - u}{v - u}\right) \left(\frac{I - u}{1 + u}\right) \zeta + \left(\frac{u}{I + u}\right) \zeta + \left(\frac{u}{I - u}\right) \zeta + (1) V \left(\frac{\zeta}{1 + u}\right) =
\]

\[
\left(\frac{3 - u}{v - u}\right) \left(\frac{I - u}{1 + u}\right) \zeta + \left(\frac{I - u}{1 - u}\right) \left(\frac{v - u}{v - u}\right) \zeta + \left(\frac{u}{I + u}\right) \zeta + \left(\frac{u}{I + u}\right) \zeta + (1) V \left(\frac{\zeta}{1 + u}\right) =
\]

\[
\left(\frac{u}{I - u}\right) \zeta + (1 - u) V \left(\frac{u}{I + u}\right) = (u) V
\]

and thus
I wrote programs for several different simple sorts and ran them on 10,000 (alphabetically) random integer data items. The following was what I got.

<table>
<thead>
<tr>
<th>Items</th>
<th>1.622</th>
<th>1.495</th>
<th>1.667</th>
<th>1.694</th>
<th>2.723</th>
<th>3.484</th>
<th>4.428</th>
</tr>
</thead>
<tbody>
<tr>
<td>quicksort (median actual)</td>
<td>1.49383</td>
<td>1.3453</td>
<td>1.2353</td>
<td>1.1593</td>
<td>1.253</td>
<td>2.5083</td>
<td>4.995</td>
</tr>
</tbody>
</table>
This is one reason that quicksort is so popular. You don’t have to be all that great in picking a partition.

\[ \sqrt{\log n} \] (number of levels of recursion times cost per level).

Note that the cost at each level is always \( cn \). We will have \( \sqrt{\log n} \) or \( \log n \) levels, and hence the cost is

\[
[ u \cdot n + (u - 1) \cdot L + u \cdot (t - 1) \cdot L ] + [ u \cdot (t + 1) + (u \cdot t - 1) \cdot L + u \cdot (t - 1) \cdot L ] = \]

\[
[ (u \cdot L + (u \cdot t - 1) \cdot L + (u \cdot (t - 1) \cdot L)] = (u) \cdot L
\]

Proof. We have the recursion

then the worst case running time is still \( O(n \log n) \).

\[ \text{Theorem 5.9. If the partitioning always produces two sublists whose sizes have a constant ratio } r > 1, \]

A Somewhat Counterintuitive Theorem
So the lower bound for sorting by comparing keys is $\lceil \log n \rceil$. The lower bound for number of comparisons is the lower bound of the height of the decision tree.

So a decision tree, with $n$ leaves, must have height at least $\lceil \log n \rceil$. That is, $h \geq \lceil \log n \rceil$. For a binary tree of $n$ leaves and height $h$ we have $h \geq \log_2 n$.

There are $n!$ total permutations of $n$ elements, so a decision tree for sorting $n$ elements must have at least $n!$ leaves.

Below, consider an algorithm to sort three elements as a decision tree, as in the tree:

6.1. Lower Bounds in General
Proposition 6.1. The sum of all path lengths to leaves is minimized when the tree is completely balanced.

Proof. Disconnecting any subtree from a balanced binary tree, and moving that subtree so as to connect it somewhere else in a binary tree, necessarily increases the sum of the path lengths.

Consider the average case. That is, 

The average case requires looking at the average path length. 

The worst case requires looking for the maximal path down a decision tree.
\[ (\ell(u) \in \mathcal{L}) \cdot (\ell(u)) \]

**Theorem 6.2.** The average case of any sort by comparison of keys is at least

\[ \ell(u) \in \mathcal{L} \]

\[ (\ell(u) \in \mathcal{L}) \cdot (\ell(u)) \]

so the average path length is bounded below by \( \ell(u) \in \mathcal{L} \cdot (\ell(u)) \)
Bucket sort, as presented in the text, permits something like the counting sort to be run on data that doesn't necessarily have to be just integers in a fixed range.

Note that the algorithm as presented in the text uses what is in fact a prefix sum calculation in the middle. If we have the additional information that the n elements to be sorted are in the range 0 ≤ x ≤ k for some known value k, then we can sort in time $\Theta(n)$ using the counting sort.

These are all variations on relatively obvious ideas; we will not dwell on these algorithms. Since all the following take time less than $n \log n$, none of these can do a sort based on comparing elements.
Heapsort is definitely not stable.

For example, merge sort is stable.

with Sort(a) appearing before Sort(b) in the resultant array.

**Definition 7.1.** A sort is **stable** if two data items a and b with equal keys and i > j always terminates significant, and we use a stable sort in every pass.

In general, we can do more than one bit at a time provided we sort on digits least significant to most.

In q passes, the elements are sorted.

If we have q-bit numbers, then we can sort on least significant bit, then next least significant, and so forth.
We can do better, indeed $3n/2$, by initializing the CREW PRAM algorithm and by using some comparisons.

Find the min, taking $n - 2$ further comparisons.
The totally naive, and slow, method would be to find the max first, taking $n - 1$ comparisons, and then to

so this bound is sharp.

Proof. First we comment that it is possible to do both max and min simultaneously in $3n/2 - 2$ comparisons.

at least $3n/2$ - 2 comparisons in the worst case.

Theorem 8.3. Any algorithm to find the max and min of $n$ elements by comparison of keys must make

Max and Min

8.1 Middle two elements.

For $n$ even, we fudge and use the average of the

For $n - 1/2$ are smaller and $(n - 1)/2$ are larger. (For $n$ even, we fudge and use the average of the

Definition 8.2. The median element in a list of $n$ elements for $n$ odd is that element for which

smallest item for any chosen $i$.

Definition 8.1. The selection problem is the problem of finding, in an array of $n$ items, the $i$-th

Selection Algorithms
We aren't going to go the effort of proving the last extra minus 2.

Compare extra beyond those needed for the max.

\[ 1 - \frac{8}{u} = \ldots + \frac{8}{u} + \frac{4}{u} + \ldots + \frac{1}{u} \]

Now go back and use the \( \frac{n}{u} \) smaller elements of the pairs from the first step, and find the \( \frac{u}{4} \) smaller steps.

Compare the winners to get \( \frac{n}{4} \) new winners. Continue until we get the max in \( \frac{n}{2} \) winners. Compare the odd subscripts elements (say) with the even subscripts elements to get \( \frac{n}{2} \) winners.
comparisons. Actually, \( u + \lceil \log_2 u \rceil - 2 \)

lost to \( x_4 \), namely, \( x_0, x_7, x_9 \). In general, then, we can find the second largest element in \( u + \lceil \log_2 u \rceil - 2 \)

We have \( x_4 \) as the max. The only possible candidates for the second largest element are those that

We would like to do better. We can clearly find the second largest key by doing two passes of max and taking

\( 2 - (u - 1) - (u - 2) \) comparisons.
Theorem 8.6. The \( i \)-th smallest element in a list of \( n \) items can be found in worst case time that is \( O(n) \).

8.3 Selection in Worst-Case Linear Time

Proof. We won't prove this.

Comparison of keys must make at least \( n \log n - 2 \) comparisons.

Theorem 8.5. In the worst case, an algorithm that finds the second largest element of \( n \) keys by

Comparison of keys needs \( n \log n - 2 \) comparisons.

Theorem 8.4. In the worst case, an algorithm that finds the second largest element of \( n \) keys by

This proves the following theorem.
the number of keys larger than and the number smaller than the median will be balanced.

The “adversary” strategy is as follows. Choose a particular value to be the median. The first time that a key is used in a comparison, a value will be assigned to that key. For as long as it is possible to do so,

We have to do at least \( n - 1 \) crucial comparisons in order to find the median.

**Definition 8.8.** A comparison involving a key \( x \) is a crucial comparison for a given value if it is the first comparison where \( x < \) for some \( y \), or \( x > \) for some \( y \), or \( x \geq \) median. A comparison is not crucial if \( x < \) median and \( y \geq \) median (or symmetrically).

Known could be changed so as to make the output incorrect.

We claim that any algorithm that outputs the median must also construct the information that relates all other keys to the median. Otherwise, the value of some key \( y \) whose relation relative to the median isn’t known could be changed so as to make the output incorrect.

**Proof.** We assume keys are distinct.

at least \( 3(n - 1)/2 \) comparisons in the worst case.

**Theorem 8.7.** Any algorithm to find, by comparison of keys, the median of \( n \) keys for \( n \) odd must do
Hence \((1 - u) + \frac{1}{2}\) total.

We can force the algorithm to make \((1 - u) + \frac{1}{2}\) nonrecursive comparisons using the above table.

None of these are nonrecursive.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign a value larger than median to the N Key</td>
<td>S, N</td>
</tr>
<tr>
<td>Assign a value larger than median to the N Key</td>
<td>N, S</td>
</tr>
<tr>
<td>Assign a value smaller than median to the N Key</td>
<td>T, N</td>
</tr>
<tr>
<td>Assign a value smaller than median to the N Key</td>
<td>N, T</td>
</tr>
<tr>
<td>One smaller, one larger</td>
<td>N, N</td>
</tr>
<tr>
<td>Adversary strategy</td>
<td>Case</td>
</tr>
</tbody>
</table>
Dynamic Data Structures

1. Why do we need dynamic data structures?
2. Fixed size, but much smaller than the full data set
3. Expandable in increments
4. Array doubling
5. Size of the data structure
6. Data that isn't known about in advance
7. Online data rather than static data
8. Symbol tables
9. Large data sets to be searched
Note: Most of these operations pass a key value in and expect a pointer returned.

- Predecessor (of an element in a set)
- Successor (of an element in a set)
- Minimum (element in a set)
- Maximum (element in a set)
- Delete (an element from a set)
- Insert (an element into a set)
- Search (for a key value in a set)
9.3.2 Good Hash Functions

9.3.1 Basic Issues in Hashing

Distinct from, but not unrelated to, hashing for cryptography:
- Good for presence/absent tests
- Good for inverting large sets of keys without sorting
- If the keys are clustered, hashing can randomize the effective key
- If the key is large relative to the number of data items, hashing can be used

Multiplication function: $f(x) = (x \mod m) \cdot \left(a \cdot (x \mod m) + b \cdot (x \mod m) \right)$
9.4 References

\[
\left( \left( \frac{y}{\mod m} \right)^2 \right) + 1 = \left( \frac{y}{\mod m} \right)^3
\]

and

\[
\left( \frac{y}{\mod m} \right)^2 = \left( \frac{y}{\mod m} \right)
\]

less than \( m \).

Double Hashing: \( y \mod (m + 1) = \frac{y}{y} \mod m \) for the \( i \)-th probe. Perhaps \( m \) prime, \( n \) slightly

Quadratic Probing: \( y \mod (m + 1) = \frac{y}{y} \mod m \) for the \( i \)-th probe

Linear Probing: \( y \mod \) for the \( i \)-th probe

How to deal with collisions? What's the probe sequence?
If it would be nice to be able to create balanced trees in an online situation, but we can’t always do that.

Definition 9.1. A binary tree in which the nodes have keys from an ordered set has the binary search tree property if the key at each node is greater than all the keys in its left subtree and less than or equal to all keys in its right subtree. In this case the tree is called a binary search tree.
Note that if we have a binary search tree, then we can print the elements in order by doing an inorder traversal.

- Print current key
- Inorder traversal
- Preorder traversal
- Postorder traversal
- Inorder traversal
- Preorder traversal
- Postorder traversal
- Inorder traversal
Balanced binary trees are dearly great, but if we had to do dynamic insertion, we could wind up with this.
```c
{ 
  { 
    return (tree-search(right[x], k));
  } 
  else 
  { 
    return (tree-search(left[x], k));
  } 
  if (k > key[x]) 
  { 
    return (x); //
  } 
  else 
  { 
    return (null); //
  } 
}
```
void tree-search(x, k)
{
    x = root[x];
    while((x != NIL) && (k != key[x]))
    {
        if(k < key[x])
            x = left[x];
        else
            x = right[x];
    }
    return(x);
}
{ return
    { 
      [right][x] = x
    }
} void tree-maximum(x) { MAXIMUM //

{ return
    { 
      [left][x] = x
    }
} void tree-minimum(x) { MINIMUM //

void tree-successor(x) {
    //
    return
    {
        [^] parent = ^
            {^ x = x
            }
        if [^] right = NIL (x) \x { [^] parent = ^
            [^] parent = [x]
            }
        else
            {
                return(tree-minimum(right[x])
            )
        )
    return
    ^}
    }
else
{
    z = [L]root
}

if NILL == \n
    \n
else
{

    if \n
        \n
else
{

    if \n
        \n
        \n
    (NILL == x)

    \n
    \n
    \n
    \n
    (x == NIL)

    \n
    \n
    \n
void tree-insert(T, z, NIL, NIL, NIL, \n\n\nINERTION node z, key, right, left) //
{ }

{ }

;'z = \forall [z]

else

{ }

;'z = \exists [z]

if (key > [key] [z])