Problem of the Day

INPUT: a set of $2n$ numbers.
WANT: partition into n pairs, so that the partition minimizes the maximum sum of a pair.

Problem of the Day

INPUT: a set of $2n$ numbers.
WANT: partition into n pairs, so that the partition minimizes the maximum sum of a pair.
Example: (1,3,5,9).

Problem of the Day

INPUT: a set of $2n$ numbers.
WANT: partition into n pairs, so that the partition minimizes the maximum sum of a pair.
Example: (1,3,5,9).
The possible partitions are

1. ((1,3),(5,9)): sum $=$ (4,14). max $= 14$.

Problem of the Day

INPUT: a set of $2n$ numbers.
WANT: partition into n pairs, so that the partition minimizes the maximum sum of a pair.
Example: (1,3,5,9).
The possible partitions are

1. ((1,3),(5,9)): sum $=$ (4,14). max $= 14$.
2. ((1,5),(3,9)): sum $=$ (6,12). max $= 12$.

Problem of the Day

INPUT: a set of $2n$ numbers.
WANT: partition into n pairs, so that the partition minimizes the maximum sum of a pair.
Example: (1,3,5,9).
The possible partitions are

1. ((1,3),(5,9)): sum $=$ (4,14). max $=$ 14.
2. ((1,5),(3,9)): sum $=$ (6,12). max $=$ 12.
3. ((1,9),(3,5)): sum $=$ (10,8). max $=$ 10

Problem of the Day

INPUT: a set of $2n$ numbers.
WANT: partition into n pairs, so that the partition minimizes the maximum sum of a pair.
Example: (1,3,5,9).
The possible partitions are

1. ((1,3),(5,9)): sum $=$ (4,14). max $=$ 14.
2. ((1,5),(3,9)): sum $=$ (6,12). max $=$ 12.
3. ((1,9),(3,5)): sum $=$ (10,8). max $=$ 10

Problem of the Day

INPUT: a set of $2n$ numbers.
WANT: partition into n pairs, so that the partition minimizes the maximum sum of a pair.
Example: (1,3,5,9).
The possible partitions are

1. ((1,3),(5,9)): sum $=$ (4,14). max $= 14$.
2. ((1,5),(3,9)): sum $=$ (6,12). max $= 12$.
3. ((1,9),(3,5)): sum $=$ (10,8). max $= 10$

Third one has the minimum max!

Solution

Importance of Sorting

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

1. Computers spend more time sorting than anything else, historically 25% on mainframes.

Importance of Sorting

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

1. Computers spend more time sorting than anything else, historically 25% on mainframes.
2. Sorting is the best studied problem in CS, with a variety of different algorithms.

Importance of Sorting

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

1. Computers spend more time sorting than anything else, historically 25% on mainframes.
2. Sorting is the best studied problem in CS, with a variety of different algorithms.
3. Most of the interesting ideas in algs can be taught in the context of sorting: divide & conquer, randomized algs, & lower bounds.

Importance of Sorting

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

1. Computers spend more time sorting than anything else, historically 25% on mainframes.
2. Sorting is the best studied problem in CS, with a variety of different algorithms.
3. Most of the interesting ideas in algs can be taught in the context of sorting: divide & conquer, randomized algs, & lower bounds.

Importance of Sorting

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

1. Computers spend more time sorting than anything else, historically 25% on mainframes.
2. Sorting is the best studied problem in CS, with a variety of different algorithms.
3. Most of the interesting ideas in algs can be taught in the context of sorting: divide & conquer, randomized algs, & lower bounds.

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

Efficiency of Sorting

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

Efficiency of Sorting

Sorting is important because once a set of items is sorted, many other problems become easy. Using $O(n \log n)$ sorting algorithms leads naturally to sub-quadratic algorithms for these problems.

Efficiency of Sorting

Sorting is important because once a set of items is sorted, many other problems become easy. Using $O(n \log n)$ sorting algorithms leads naturally to sub-quadratic algorithms for these problems.

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

Efficiency of Sorting

Sorting is important because once a set of items is sorted, many other problems become easy. Using $O(n \log n)$ sorting algorithms leads naturally to sub-quadratic algorithms for these problems.

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

Large-scale data processing would be impossible if sorting took $\Omega(n^2)$ time.

Application of Sorting: Searching

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

Application of Sorting: Searching

- Binary search lets you test whether an item is in a dictionary in $O(\lg n)$ time.
- Search preprocessing is perhaps the single most important application of sorting.

Application of Sorting: Closest pair

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

Application of Sorting: Closest pair

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

Application of Sorting: Element Uniqueness

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

Application of Sorting: Element Uniqueness

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

- Sort them and do a linear scan to check all adjacent pairs.
- This is a special case of closest pair above.

Application of Sorting: Mode

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

Application of Sorting: Mode

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

- Sort them and do a linear scan to measure the length of all adjacent runs.
- The number of instances of k in a sorted array can be found in $O(\log n)$ time by using binary search to look for the positions of both $k - \epsilon$ and $k + \epsilon$.

Application of Sorting: Median and Selection

What is the kth smallest item in the set?

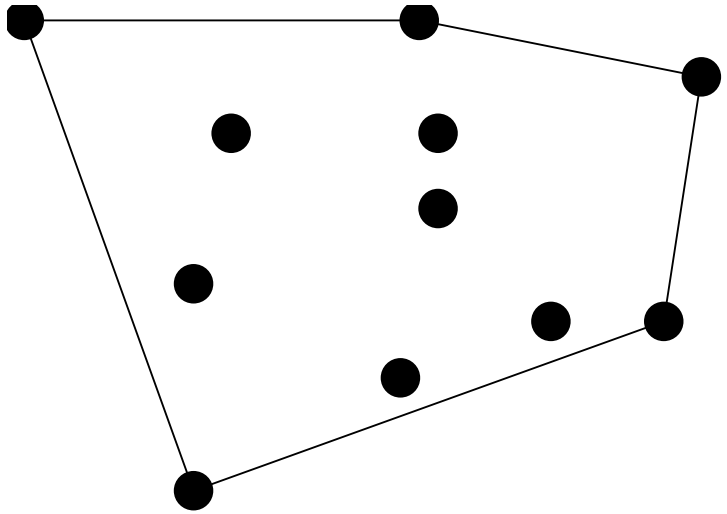
Application of Sorting: Median and Selection

What is the kth smallest item in the set?

- Once the keys are placed in sorted order in an array, the kth smallest can be found in constant time by simply looking in the kth position of the array.
  comes from partial sorting.

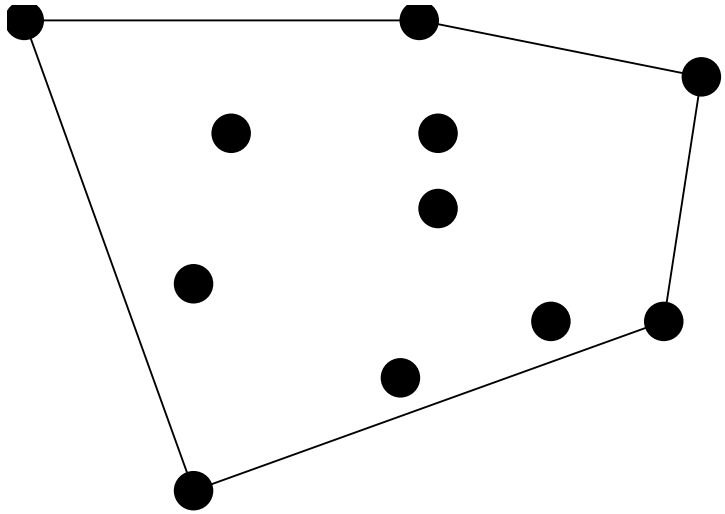
Application of Sorting: Convex hulls

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



Application of Sorting: Convex hulls

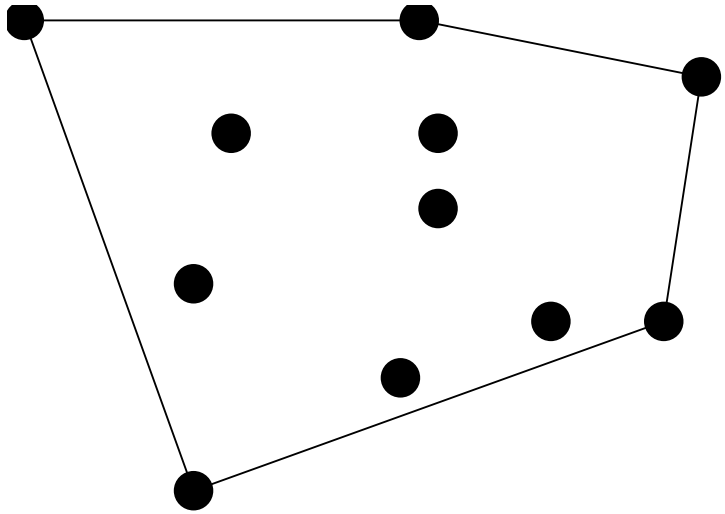
Given n points in two dimensions, find the smallest area convex polygon which contains them all.



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

Application of Sorting: Convex hulls

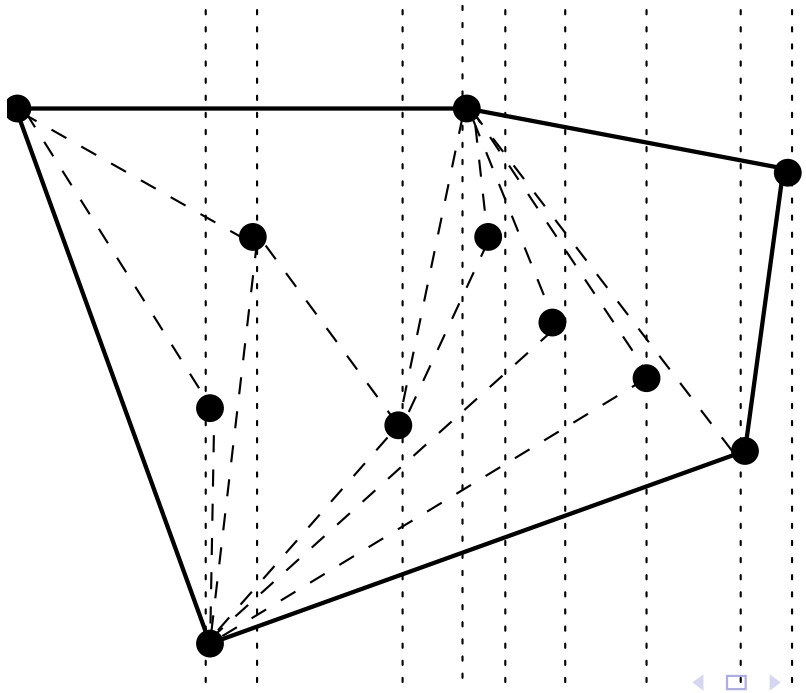
Given n points in two dimensions, find the smallest area convex polygon which contains them all.



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

Finding Convex Hulls

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



Pragmatics of Sorting: Comparison Functions

- Alphabetizing is the sorting of text strings.

Pragmatics of Sorting: Comparison Functions

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

Pragmatics of Sorting: Comparison Functions

- Alphabetizing is the sorting of text strings.
- Libraries have very complete and complicated rules concerning the relative *collating sequence* of characters and punctuation.
- Is *Skiena* the same key as *skiena*?

Pragmatics of Sorting: Comparison Functions

- Alphabetizing is the sorting of text strings.
- Libraries have very complete and complicated rules concerning the relative *collating sequence* of characters and punctuation.
- Is *Skiena* the same key as *skiena*?
- Is *Brown-Williams* before or after *Brown America* before or after *Brown, John*?

Pragmatics of Sorting: Comparison Functions

- Alphabetizing is the sorting of text strings.
- Libraries have very complete and complicated rules concerning the relative *collating sequence* of characters and punctuation.
- Is *Skiena* the same key as *skiena*?
- Is *Brown-Williams* before or after *Brown America* before or after *Brown, John*?
- Explicitly controlling the order of keys is the job of the *comparison function* we apply to each pair of elements.

Pragmatics of Sorting: Equal Elements

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

Pragmatics of Sorting: Equal Elements

- Elements with equal key values will all bunch together in any total order, but sometimes the relative order among these keys matters.
- Sorting algorithms that always leave equal items in the same relative order as in the original permutation are called *stable*.

Pragmatics of Sorting: Equal Elements

- Elements with equal key values will all bunch together in any total order, but sometimes the relative order among these keys matters.
- Sorting algorithms that always leave equal items in the same relative order as in the original permutation are called *stable*.
- Unfortunately very few fast algorithms are stable, but stability can be achieved by adding the initial position as a secondary key.

Pragmatics of Sorting: Library Functions

- Any reasonable programming language has a built-in sort routine as a library function.
- You are almost always better off using the system sort than writing your own routine.

Pragmatics of Sorting: Library Functions

- Any reasonable programming language has a built-in sort routine as a library function.
- You are almost always better off using the system sort than writing your own routine.
- For example, the standard library for C contains the function `qsort` for sorting:

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

Selection Sort

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

For $i = 1$ to n
A: Find the smallest of the first $n - i + 1$ items.
B: Pull it out of the array and put it first.

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

The Data Structure Matters

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

The Data Structure Matters

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

Using balanced search trees or heaps, both of these operations can be done within $O(\lg n)$ time, for an $O(n \log n)$ selection sort, balancing the work and achieving a better tradeoff.

Heap Definition

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

1. All leaves are on, at most, two adjacent levels.

Heap Definition

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

1. All leaves are on, at most, two adjacent levels.
2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.

Heap Definition

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

1. All leaves are on, at most, two adjacent levels.
2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
3. The key in any node is $\leq$ keys in its children.

Heap Definition

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

1. All leaves are on, at most, two adjacent levels.
2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
3. The key in any node is $\leq$ keys in its children.

Heap Definition

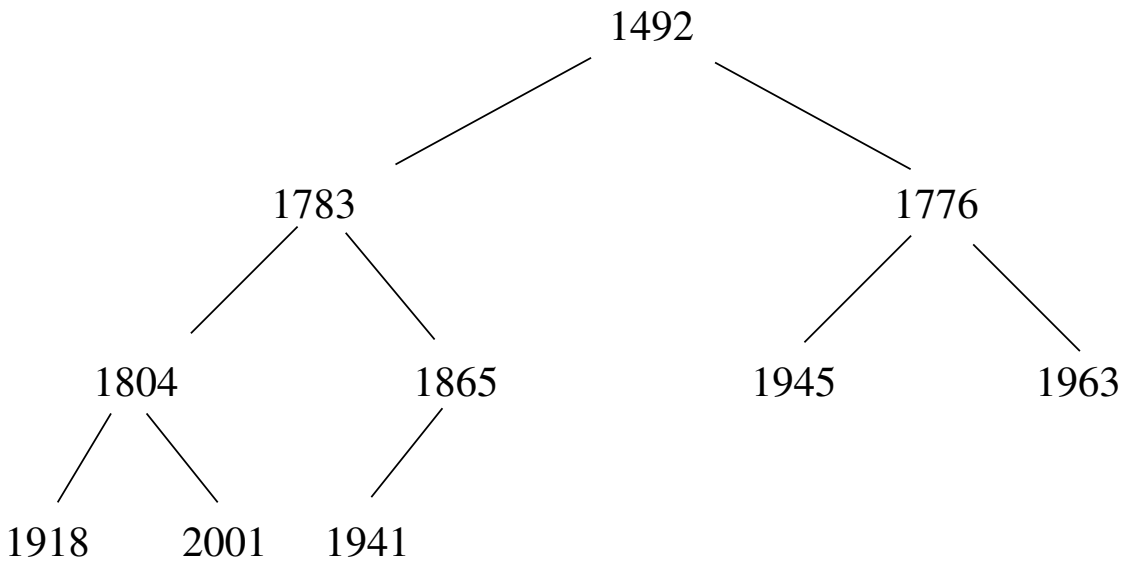
A binary heap is defined to be a binary tree with a key in each node such that:

1. All leaves are on, at most, two adjacent levels.
2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
3. The key in any node is $\leq$ keys in its children.

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

Binary Heaps

Heaps maintain a partial order on the set of elements which is weaker than the sorted order (so it's more efficient to maintain).



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

Partial Orders

The ancestor relation in a heap defines a *partial order*:

1. *Reflexive:* x is an ancestor of itself.

Partial Orders

The ancestor relation in a heap defines a *partial order*:

1. *Reflexive:* x is an ancestor of itself.
2. *Anti-symmetric:* if x is an ancestor of y and y is an ancestor of x, then $x = y$.

Partial Orders

The ancestor relation in a heap defines a *partial order*:

1. *Reflexive:* x is an ancestor of itself.
2. *Anti-symmetric:* if x is an ancestor of y and y is an ancestor of x, then $x = y$.
3. *Transitive:* if x is an ancestor of y and y is an ancestor of z, x is an ancestor of z.

Partial Orders

The ancestor relation in a heap defines a *partial order*:

1. *Reflexive:* x is an ancestor of itself.
2. *Anti-symmetric:* if x is an ancestor of y and y is an ancestor of x, then $x = y$.
3. *Transitive:* if x is an ancestor of y and y is an ancestor of z, x is an ancestor of z.

Partial Orders

The ancestor relation in a heap defines a *partial order*:

1. *Reflexive:* x is an ancestor of itself.
2. *Anti-symmetric:* if x is an ancestor of y and y is an ancestor of x, then $x = y$.
3. *Transitive:* if x is an ancestor of y and y is an ancestor of z, x is an ancestor of z.

Partial orders can be used to model hierarchies with incomplete information or equal-valued elements.

Why Heaps?

The partial order defined by the heap structure is weaker than that of the total order, which explains

1. Why it is easier to build.
2. Why it is less useful than sorting (but still very important).

Array-Based Heaps

- We would normally use pointers to represent a binary tree.

Array-Based Heaps

- We would normally use pointers to represent a binary tree.
- However, we store a heap as an array of keys, using the position of the keys to *implicitly* satisfy the role of the pointers.

Array-Based Heaps

- We would normally use pointers to represent a binary tree.
- However, we store a heap as an array of keys, using the position of the keys to *implicitly* satisfy the role of the pointers.
- The *left* child of k sits in position $2k$ and the right child in $2k + 1$.

Array-Based Heaps

- We would normally use pointers to represent a binary tree.
- However, we store a heap as an array of keys, using the position of the keys to *implicitly* satisfy the role of the pointers.
- The *left* child of k sits in position $2k$ and the right child in $2k + 1$.
- The parent of k is in position $\lfloor k/2 \rfloor$.

Array-Based Heaps

- We would normally use pointers to represent a binary tree.
- However, we store a heap as an array of keys, using the position of the keys to *implicitly* satisfy the role of the pointers.
- The *left* child of k sits in position $2k$ and the right child in $2k + 1$.
- The parent of k is in position $\lfloor k/2 \rfloor$.
- grandparent?

Can we Implicitly Represent Any Binary Tree?

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

Can we Implicitly Represent Any Binary Tree?

- The implicit representation is inefficient if the tree is sparse, meaning that the number of nodes $n < 2^h$.
- All missing internal nodes still take up space in our structure.

Can we Implicitly Represent Any Binary Tree?

- The implicit representation is inefficient if the tree is sparse, meaning that the number of nodes $n < 2^h$.
- All missing internal nodes still take up space in our structure.
- This is why we insist on heaps as being as balanced/full at each level as possible.

Heap's Height

A heap with n elements has height $h = \lfloor \lg n \rfloor$.

Heap Insertion

1. Put new element in $(n+1)$st location in the array.
2. "Bubble" it up to the correct place.

Constructing Heaps by Insertion

Heaps can be constructed incrementally, by inserting new elements into the heap.

Constructing Heaps by Insertion

Heaps can be constructed incrementally, by inserting new elements into the heap.
Doing n such insertions takes $\Theta(n \log n)$.

Constructing Heaps by Insertion

Heaps can be constructed incrementally, by inserting new elements into the heap.
Doing n such insertions takes $\Theta(n \log n)$.
$\log 1 + \log 2 + \ldots \log n$

Bubble Down or Heapify

Robert Floyd found a better way to build a heap, using a procedure called heapify.
Given two heaps and a fresh element, they can be merged into one by making the new one the root and bubbling down.

HEAPIFY(A, i)

Makes the subtree rooted at i a heap provided that the left and right subtrees of i are already heaps.

1. Let *min* be the smaller of i's two children: $\min(A[2i], A[2i+1])$
2. **if** $A[i] < min$
   then "we're done"
   else
   Swap $A[i]$ with *min*
   HEAPIFY($A, 2i$ "or" $2i+1$)

Time Complexity of Heapify

$O(1)$ effort at each level $\times$ height $= O(\log n)$.

BUILD_HEAP

Given an arbitrary array, convert it into a heap.

Build-heap(A)

$n = |A|$

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

Heapify(A,i)

Time Complexity (simple analysis) : $O(n)$ calls to HEAPIFY, each call is $O(\log n)$, So $O(n \log n)$.

Is our Analysis Tight?

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

Is our Analysis Tight?

"Are we doing a careful analysis?
Might our algorithm be faster than it seems?"
Doing at most x operations of at most y time each takes total time $O(xy)$.

Is our Analysis Tight?

"Are we doing a careful analysis?
Might our algorithm be faster than it seems?"
Doing at most x operations of at most y time each takes total time $O(xy)$.
However, if we overestimate too much, our bound may not be as tight as it should be!

Exact Analysis

BuildHeap performs better than $O(n \log n)$, because most of the heaps we merge are extremely small.

$$\begin{aligned} T(n) &= \frac{n}{2} \times 1 + \frac{n}{4} \times 2 + \frac{n}{8} \times 3 + \ldots + 1 \times \log n \\ &= \sum_{i=1}^{\log n} \frac{n}{2^i} i \\ &\leq n \sum_{i=0}^{\infty} \frac{i}{2^i} \end{aligned}$$

Not quite a geometric series.

Proof of Convergence (*)

The identity for the sum of a geometric series is

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

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

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

Proof of Convergence 2 (*)

Multiplying both sides of the equation by x gives:

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

Substituting $x = 1/2$ gives 2, so Build-heap takes $2n = O(n)$ time.

Loose Ends

Min-Heap v/s Max-Heap
Deletion of Max Element from Max Heap

Heapsort (uses max-heap)

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

Exchanging the max element with the last element and calling heapify repeatedly gives an $O(n \lg n)$ sorting algorithm.

Heapsort Demo

Priority Queues

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

Priority Queues

- Priority queues are data structures which provide extra flexibility over sorting.
- This is important because jobs often enter a system at arbitrary times.

Priority Queues

- Priority queues are data structures which provide extra flexibility over sorting.
- This is important because jobs often enter a system at arbitrary times.
- It is more cost-effective to insert a new job into a priority queue than to re-sort everything on each new arrival.

(Max) Priority Queue Operations

- *Insert(Q,x)*: Given an item x with key k, insert it into priority queue Q.
- *Find-Maximum(Q)*: Return a ptr to item with max key value in priority queue Q.
- *Delete-Maximum(Q)* – Remove the item from priority queue Q whose key is maximum.

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

Applications of Priority Queues: Discrete Event Simulations

- In simulations of airports, parking lots, and computer networks – priority queues can be used to maintain who goes next.

Applications of Priority Queues: Discrete Event Simulations

- In simulations of airports, parking lots, and computer networks – priority queues can be used to maintain who goes next.
- The stack and queue orders are just ideal cases of orderings.

Applications of Priority Queues: Discrete Event Simulations

- In simulations of airports, parking lots, and computer networks – priority queues can be used to maintain who goes next.
- The stack and queue orders are just ideal cases of orderings.
- In real life, people cut in line!

Data Structures Handbook

- Double-ended priority queues.
- Pages 15 and 160

Problem of the Day

Give an efficient algorithm to determine whether two sets (of size m and n) are disjoint. Analyze the complexity of your algorithm in terms of m and n.

Solution

Mergesort

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

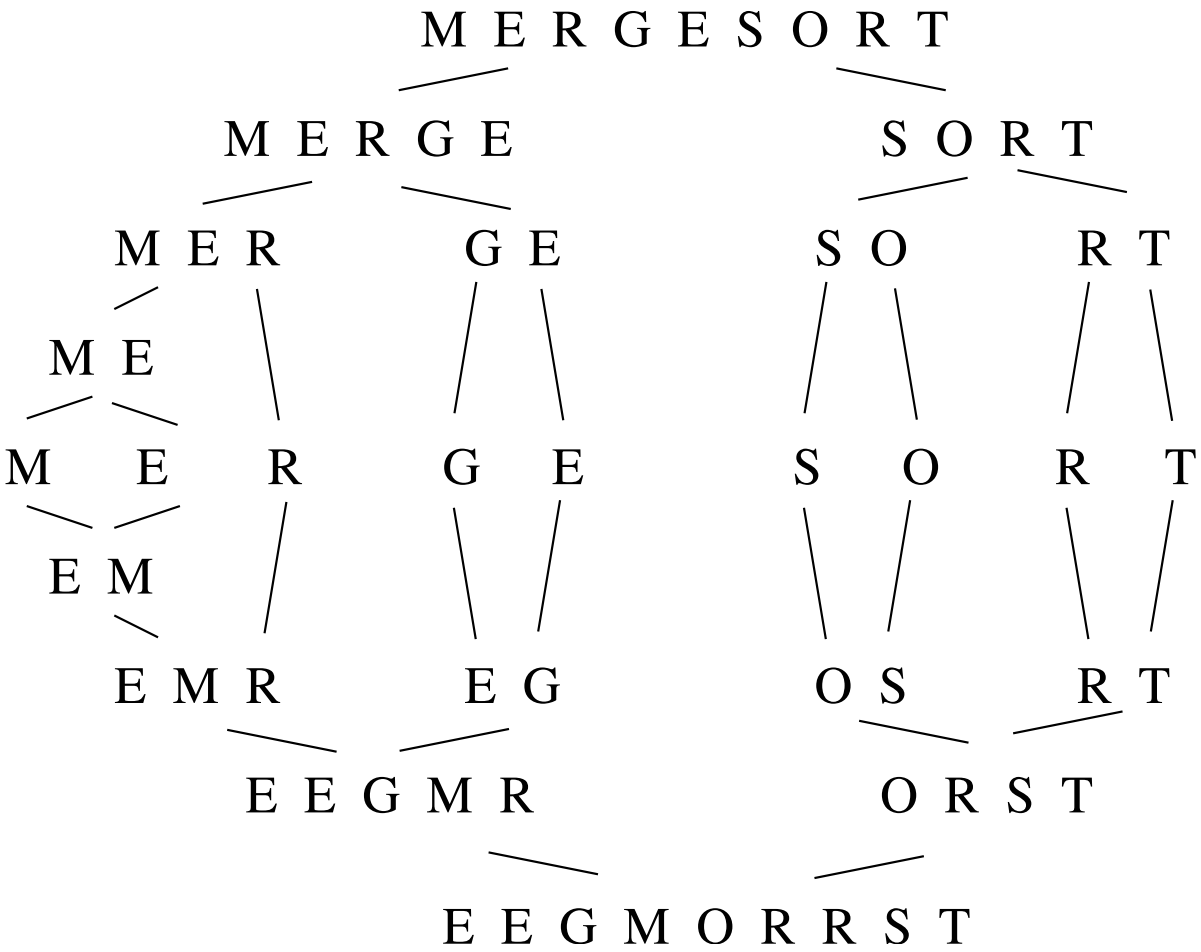
Mergesort Implementation

```
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 Example



Merging Sorted Lists

- The efficiency of mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list.

Merging Sorted Lists

- The efficiency of mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list.
- The smallest element can be removed, leaving two sorted lists behind, one slightly shorter than before.

Merging Sorted Lists

- The efficiency of mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list.
- The smallest element can be removed, leaving two sorted lists behind, one slightly shorter than before.
- Repeating this operation until both lists are empty merges two sorted lists (with a total of n elements between them) into one, using at most $n - 1$ comparisons or $O(n)$ total work

Example

$A = \{5, 7, 12, 19\}$ and $B = \{4, 6, 13, 15\}$.

Buffering

- Although mergesort is $O(n \lg n)$, it is inconvenient to implement with a single array.

Buffering

- Although mergesort is $O(n \lg n)$, it is inconvenient to implement with a single array.
- Merging $(4, 5, 6)$ and $(1, 2, 3)$ would overwrite the first three elements if they were packed in an array.

Buffering

- Although mergesort is $O(n \lg n)$, it is inconvenient to implement with a single array.
- Merging $(4, 5, 6)$ and $(1, 2, 3)$ would overwrite the first three elements if they were packed in an array.
- Writing the merged list to a buffer and recopying it uses extra space but not extra time.

External Sorting

- Which $O(n \log n)$ algorithm you use for sorting doesn't matter much until n is so big the data does not fit in memory.

External Sorting

- Which $O(n \log n)$ algorithm you use for sorting doesn't matter much until n is so big the data does not fit in memory.
- Mergesort proves to be the basis for the most efficient *external sorting* programs.

External Sorting

- Which $O(n \log n)$ algorithm you use for sorting doesn't matter much until n is so big the data does not fit in memory.
- Mergesort proves to be the basis for the most efficient *external sorting* programs.
- Disks are much slower than main memory, and benefit from algorithms that read and write data in long streams – not random access.

Divide and Conquer Examples

- Divide and conquer is an important algorithm design technique used in mergesort, binary search, FFT, and Strassen's matrix multiplication algorithm ($\Theta(n^{\log_2 7})$) .

Divide and Conquer Examples

- Divide and conquer is an important algorithm design technique used in mergesort, binary search, FFT, and Strassen's matrix multiplication algorithm ($\Theta(n^{\log_2 7})$) .
- Best known algorithm for matrix multiplication is $O(n^{2.376})$.

Divide and Conquer Strategy

- We divide the problem into two smaller subproblems, solve each recursively, and then meld the two partial solutions into one solution for the full problem.

Divide and Conquer Strategy

- We divide the problem into two smaller subproblems, solve each recursively, and then meld the two partial solutions into one solution for the full problem.
- When merging takes less time than solving the two subproblems, we get an efficient algorithm.

Quicksort

In practice, the fastest internal sorting algorithm is Quicksort, which uses partitioning about a pivot as its main idea.

Quicksort

In practice, the fastest internal sorting algorithm is Quicksort, which uses partitioning about a pivot as its main idea.

The pivot is any element in the array. It could be

- an element in a specific location (e.g., the first element)
- chosen randomly

Partitioning about the Pivot

Partitioning places all the elements less than the pivot in the left part of the array, and all elements greater than the pivot in the right part of the array. The pivot fits in the slot between them.

An example of Partitioning

Why Partition?

Partitioning takes $O(n)$ time. But what does it buy us?

Why Partition?

Partitioning takes $O(n)$ time. But what does it buy us?

1. The pivot ends up in the correct place in the total order!

Why Partition?

Partitioning takes $O(n)$ time. But what does it buy us?

1. The pivot ends up in the correct place in the total order!
2. After partitioning, no element flops to the other side of the pivot in the final sorted order.

Why Partition?

Partitioning takes $O(n)$ time. But what does it buy us?

1. The pivot ends up in the correct place in the total order!
2. After partitioning, no element flops to the other side of the pivot in the final sorted order.
3. Thus, we can sort the elements to the left of the pivot and the right of the pivot independently, giving us a recursive sorting algorithm!

Quicksort Pseudocode

```
Sort(A)
        Quicksort(A,1,n)


Quicksort(A, low, high)
        if (low < high)
                pivot-location = Partition(A,low,high)
                Quicksort(A,low, pivot-location - 1)
                Quicksort(A, pivot-location+1, high)
```

Partition Implementation (skip)

```
Partition(A,low,high)
        pivot = A[low]
        leftwall = low
        for i = low+1 to high
                if (A[i] < pivot) then
                        leftwall = leftwall+1
                        swap(A[i],A[leftwall])
        swap(A[low],A[leftwall])
```

Quicksort Song

Best Case for Quicksort

- The best case for *divide-and-conquer* algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size $n/2$.

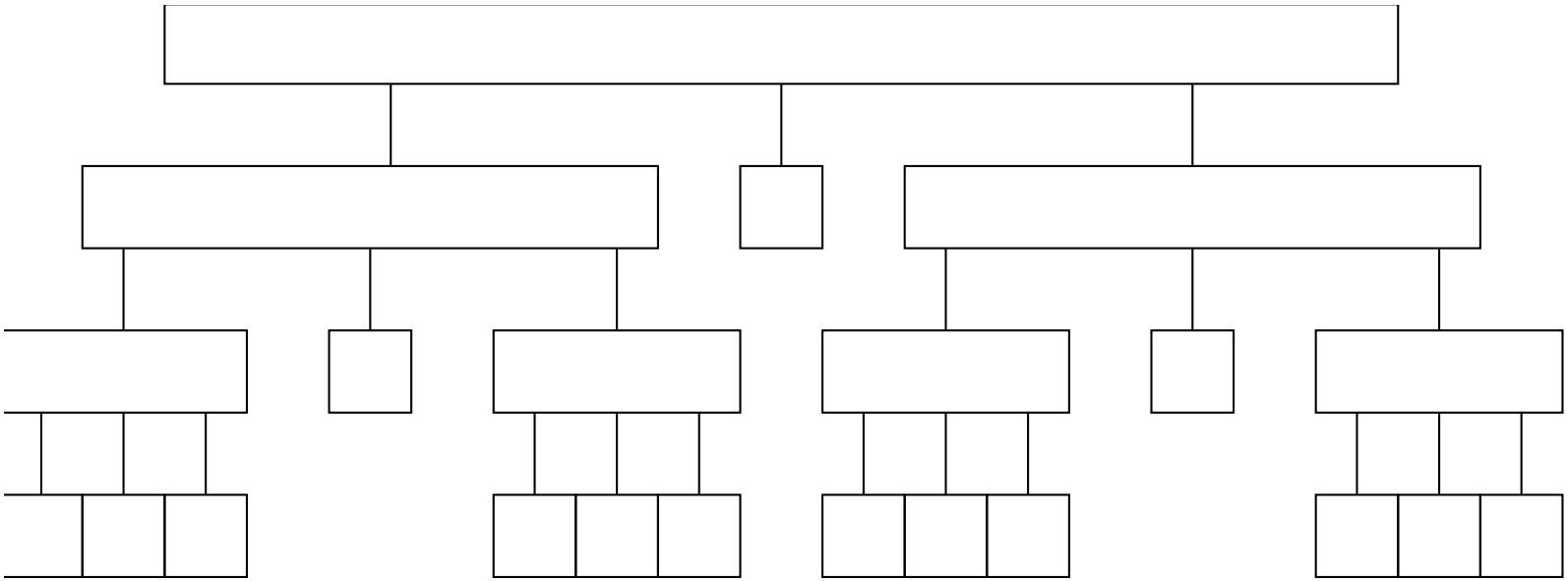
Best Case for Quicksort

- The best case for *divide-and-conquer* algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size $n/2$.
- The partition step on each subproblem is linear in its size.

Best Case for Quicksort

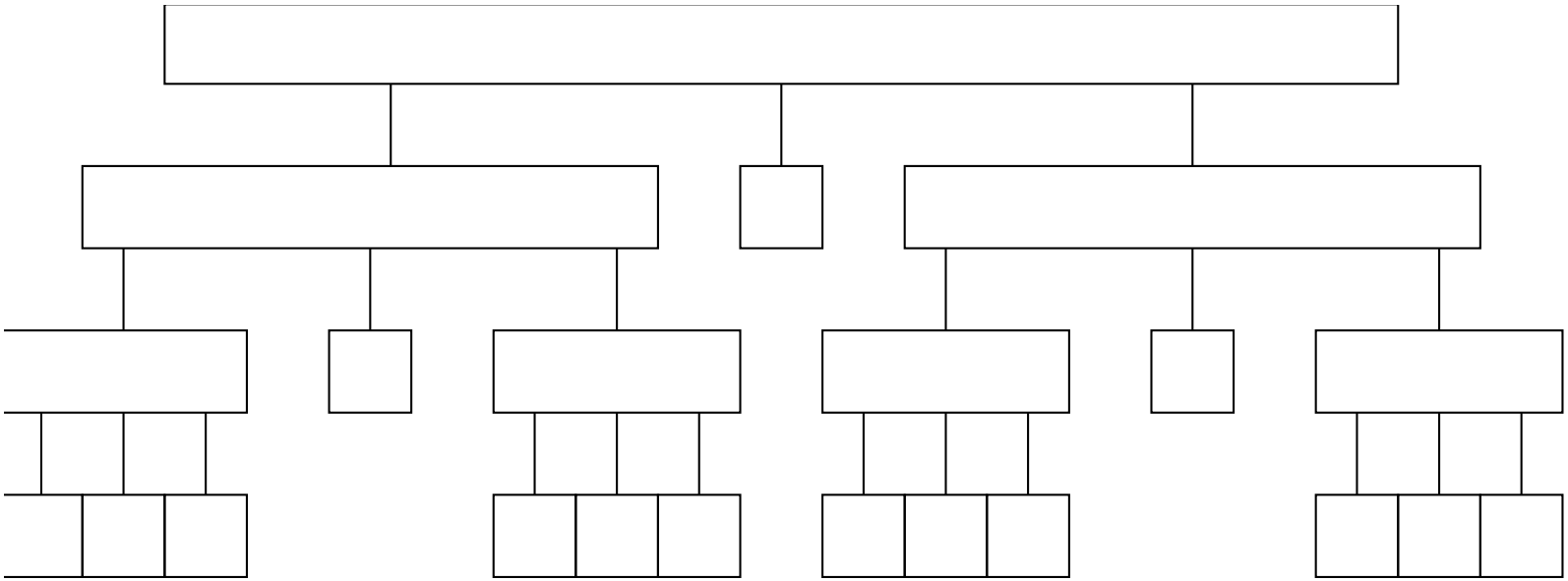
- The best case for *divide-and-conquer* algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size $n/2$.
- The partition step on each subproblem is linear in its size.
- At the kth level, the **TOTAL** effort in partitioning the 2^k problems of size $\frac{n}{2^k}$ is $O(n)$.

Best Case Recursion Tree



The partitioning on each level is $O(n)$; $\lg n$ levels.

Best Case Recursion Tree



The partitioning on each level is $O(n)$; $\lg n$ levels.
Total time in the best case is $O(n \lg n)$.

Analyzing a Recursive Algorithm

- Set up a recurrence relation; for example,

Analyzing a Recursive Algorithm

- Set up a recurrence relation; for example,
  - Let $T(n)$ represent the best case for quicksort on an input of size n.

Analyzing a Recursive Algorithm

- Set up a recurrence relation; for example,
  - Let $T(n)$ represent the best case for quicksort on an input of size n.
  - $T(n) = 2T(n/2) + n$

Analyzing a Recursive Algorithm

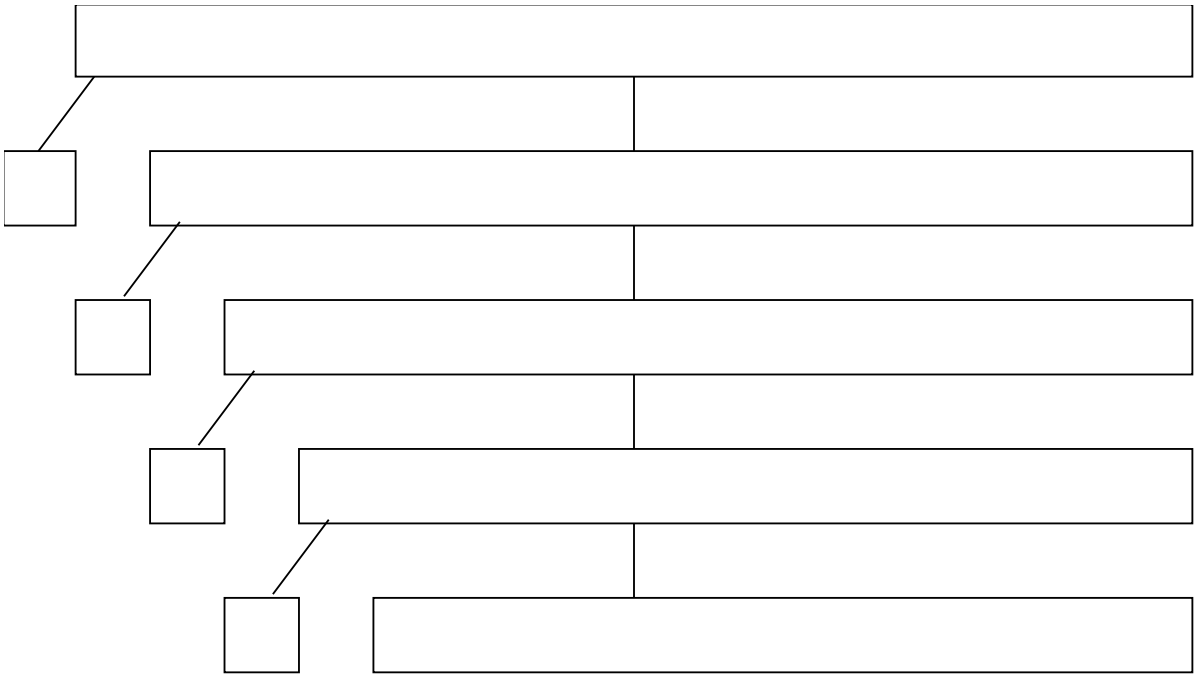
- Set up a recurrence relation; for example,
  - Let $T(n)$ represent the best case for quicksort on an input of size n.
  - $T(n) = 2T(n/2) + n$
  - Happily, the same recurrence relation works for mergesort!

Analyzing a Recursive Algorithm

- Set up a recurrence relation; for example,
  - Let $T(n)$ represent the best case for quicksort on an input of size n.
  - $T(n) = 2T(n/2) + n$
  - Happily, the same recurrence relation works for mergesort!
- Solve the recurrence relation.

Worst Case for Quicksort

Suppose instead our pivot element splits the array as unequally as possible.



Recurrence Relation for Worst Case

Now we have $n - 1$ levels, instead of $\lg n$, for a worst case time of $\Theta(n^2)$. More precisely

Recurrence Relation for Worst Case

Now we have $n-1$ levels, instead of $\lg n$, for a worst case time of $\Theta(n^2)$. More precisely

$$T(n) = T(n-1) + n$$

Recurrence Relation for Worst Case

Now we have $n-1$ levels, instead of $\lg n$, for a worst case time of $\Theta(n^2)$. More precisely

$$\begin{aligned} T(n) &= T(n-1)+n \\ &= T(n-2)+(n-1)+n \end{aligned}$$

Recurrence Relation for Worst Case

Now we have $n-1$ levels, instead of $\lg n$, for a worst case time of $\Theta(n^2)$. More precisely

$$\begin{aligned} T(n) &= T(n-1)+n \\ &= T(n-2)+(n-1)+n \\ &= \dots \\ &= T(1)+2+3+\cdots+n \end{aligned}$$

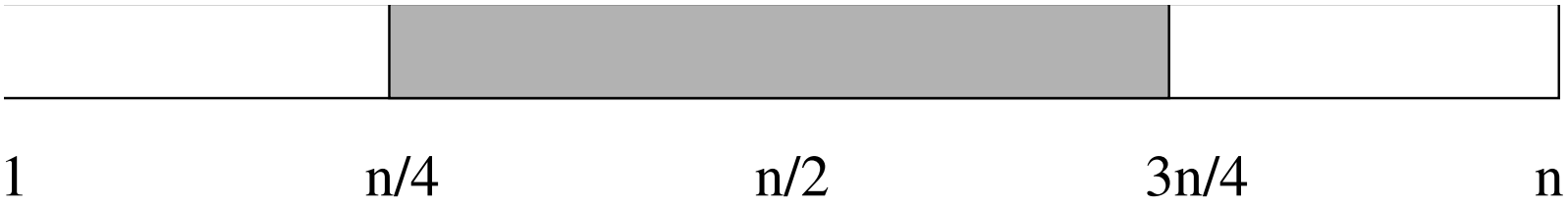
Recurrence Relation for Worst Case

Now we have $n-1$ levels, instead of $\lg n$, for a worst case time of $\Theta(n^2)$. More precisely

$$\begin{aligned} T(n) &= T(n-1)+n \\ &= T(n-2)+(n-1)+n \\ &= \dots \\ &= T(1)+2+3+\cdots+n \\ &= \sum_{i=1}^{n} i = n(n+1)/2 \end{aligned}$$

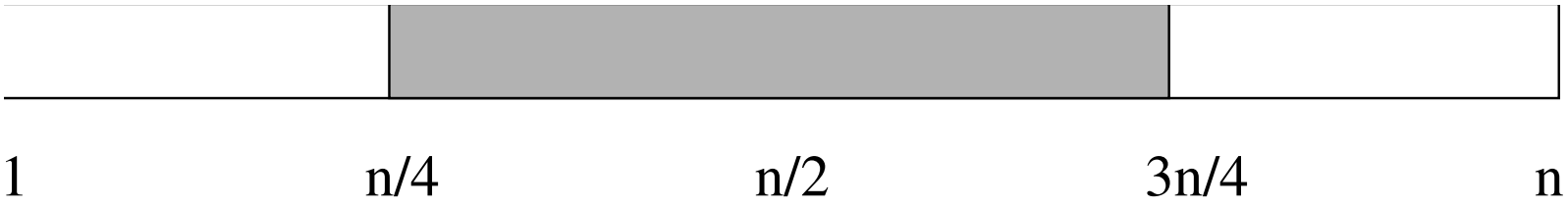
Intuition: The Average Case for Quicksort

Suppose we pick the pivot at random.



Intuition: The Average Case for Quicksort

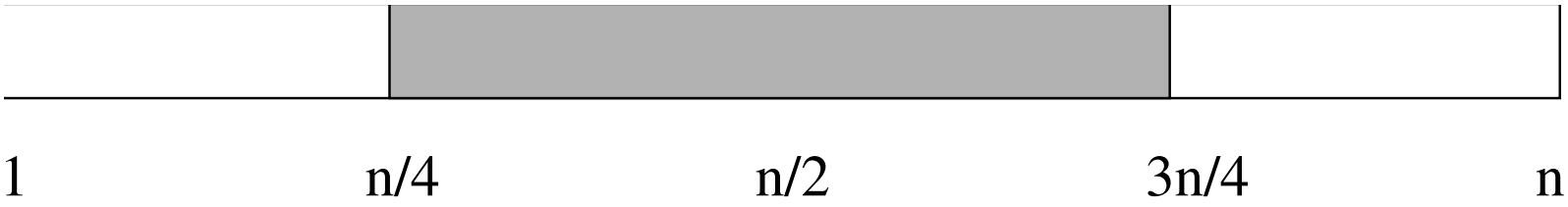
Suppose we pick the pivot at random.



Half the time, the pivot element will be from the center half of the sorted array.

Intuition: The Average Case for Quicksort

Suppose we pick the pivot at random.



Half the time, the pivot element will be from the center half of the sorted array.
Whenever the pivot element is from positions $n/4$ to $3n/4$, the larger remaining subarray contains at most $3n/4$ elements.

How Many Good Partition Levels

If we assume that the pivot element is always in this range, what is the maximum number of partition levels k we need to get from n elements down to 1 element?

$$(3/4)^k \cdot n = 1$$

How Many Good Partition Levels

If we assume that the pivot element is always in this range, what is the maximum number of partition levels k we need to get from n elements down to 1 element?

$$\begin{aligned} (3/4)^k \cdot n &= 1 \\ n &= (4/3)^k \end{aligned}$$

How Many Good Partition Levels

If we assume that the pivot element is always in this range, what is the maximum number of partition levels k we need to get from n elements down to 1 element?

$$\begin{aligned} (3/4)^k \cdot n &= 1 \\ n &= (4/3)^k \\ \lg n &= k \cdot \lg(4/3) \end{aligned}$$

How Many Good Partition Levels

If we assume that the pivot element is always in this range, what is the maximum number of partition levels k we need to get from n elements down to 1 element?

$$\begin{aligned} (3/4)^k \cdot n &= 1 \\ n &= (4/3)^k \\ \lg n &= k \cdot \lg(4/3) \\ k &= \lg(n)/\lg(4/3) \approx 2 \lg n \end{aligned}$$

How Many Bad Partitions?

Question What is the probability that a randomly chosen pivot will generate a decent partition?

How Many Bad Partitions?

Question What is the probability that a randomly chosen pivot will generate a decent partition?
Answer: Since any number ranked between $n/4$ and $3n/4$ would make a decent pivot, answer is $\frac{1}{2}$.

How Many Bad Partitions?

Question What is the probability that a randomly chosen pivot will generate a decent partition?
Answer: Since any number ranked between $n/4$ and $3n/4$ would make a decent pivot, answer is $\frac{1}{2}$.
If we need $2 \lg n$ levels of decent partitions to finish the job, and half of random partitions are decent, then on average the recursion tree to quicksort the array has $\approx 4 \lg n$ levels.

How Many Bad Partitions?

Question What is the probability that a randomly chosen pivot will generate a decent partition?
Answer: Since any number ranked between $n/4$ and $3n/4$ would make a decent pivot, answer is $\frac{1}{2}$.
If we need $2 \lg n$ levels of decent partitions to finish the job, and half of random partitions are decent, then on average the recursion tree to quicksort the array has $\approx 4 \lg n$ levels.
More careful analysis shows that the expected number of comparisons is $\approx 1.38 n \lg n$.

What *is* the Worst Case?

The worst case for Quicksort depends upon how we select our partition or pivot element.

What *is* the Worst Case?

The worst case for Quicksort depends upon how we select our partition or pivot element.
If pivot is the first element of the subarray, the worst-case occurs when the input is already sorted!
B D F H J K
D F H J K
F H J K
H J K
J K
K

Pick a Better Pivot

Having the worst case occur when they are sorted or almost sorted is *very bad*, since that is likely to be the case in certain applications.

Pick a Better Pivot

Having the worst case occur when they are sorted or almost sorted is *very bad*, since that is likely to be the case in certain applications.
To eliminate this problem, pick a better pivot:

1. Use the middle element of the subarray.
2. Take the median of three elements (first, last, middle) as the pivot.

Pick a Better Pivot

Having the worst case occur when they are sorted or almost sorted is *very bad*, since that is likely to be the case in certain applications.
To eliminate this problem, pick a better pivot:

1. Use the middle element of the subarray.
2. Take the median of three elements (first, last, middle) as the pivot.

Whichever of these rules we use, the worst case remains $O(n^2)$.

Andrew Johnson's M.S. thesis

Is Quicksort really faster than Heapsort?

- Since Heapsort is $\Theta(n \lg n)$ and selection sort is $\Theta(n^2)$, there is no debate about which will be better for decent-sized files.

Is Quicksort really faster than Heapsort?

- Since Heapsort is $\Theta(n \lg n)$ and selection sort is $\Theta(n^2)$, there is no debate about which will be better for decent-sized files.
- When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort.

Is Quicksort really faster than Heapsort?

- Since Heapsort is $\Theta(n \lg n)$ and selection sort is $\Theta(n^2)$, there is no debate about which will be better for decent-sized files.
- When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort.
- Outside the realm of asymptotic analysis.

Randomized Quicksort

Q: If you used Quicksort, what kind of data would your worst enemy (or an Andrew Johnson) give you to run it on?

Randomized Quicksort

Q: If you used Quicksort, what kind of data would your worst enemy (or an Andrew Johnson) give you to run it on?

A: Exactly the worst-case, to make you look bad.

Randomized Quicksort

Q: If you used Quicksort, what kind of data would your worst enemy (or an Andrew Johnson) give you to run it on?
A: Exactly the worst-case, to make you look bad.
But instead of picking the median of three or the first element as pivot, suppose you picked the pivot element at random.

Randomized Quicksort

Q: If you used Quicksort, what kind of data would your worst enemy (or an Andrew Johnson) give you to run it on?
A: Exactly the worst-case, to make you look bad.
But instead of picking the median of three or the first element as pivot, suppose you picked the pivot element at random.
Now your enemy can't design a worst-case instance, because no matter what they give you, you have the same probability of picking a good pivot!

Randomized Guarantees

By either picking a random pivot or scrambling the permutation before sorting it, we can say:

Randomized Guarantees

By either picking a random pivot or scrambling the permutation before sorting it, we can say:

> *"With high probability, randomized quicksort runs in* $\Theta(n \lg n)$ *time."*

Randomized Guarantees

By either picking a random pivot or scrambling the permutation before sorting it, we can say:

> *"With high probability, randomized quicksort runs in* $\Theta(n \lg n)$ *time."*

Where before, all we could say is:

> *"If you give me random input data, quicksort runs in expected* $\Theta(n \lg n)$ *time."*

Importance of Randomization

Unless we are *extremely* unlucky (as opposed to ill-prepared or unpopular) we will certainly get good performance.

Importance of Randomization

Unless we are *extremely* unlucky (as opposed to ill-prepared or unpopular) we will certainly get good performance.
Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.

Can we sort in $o(n \lg n)$?

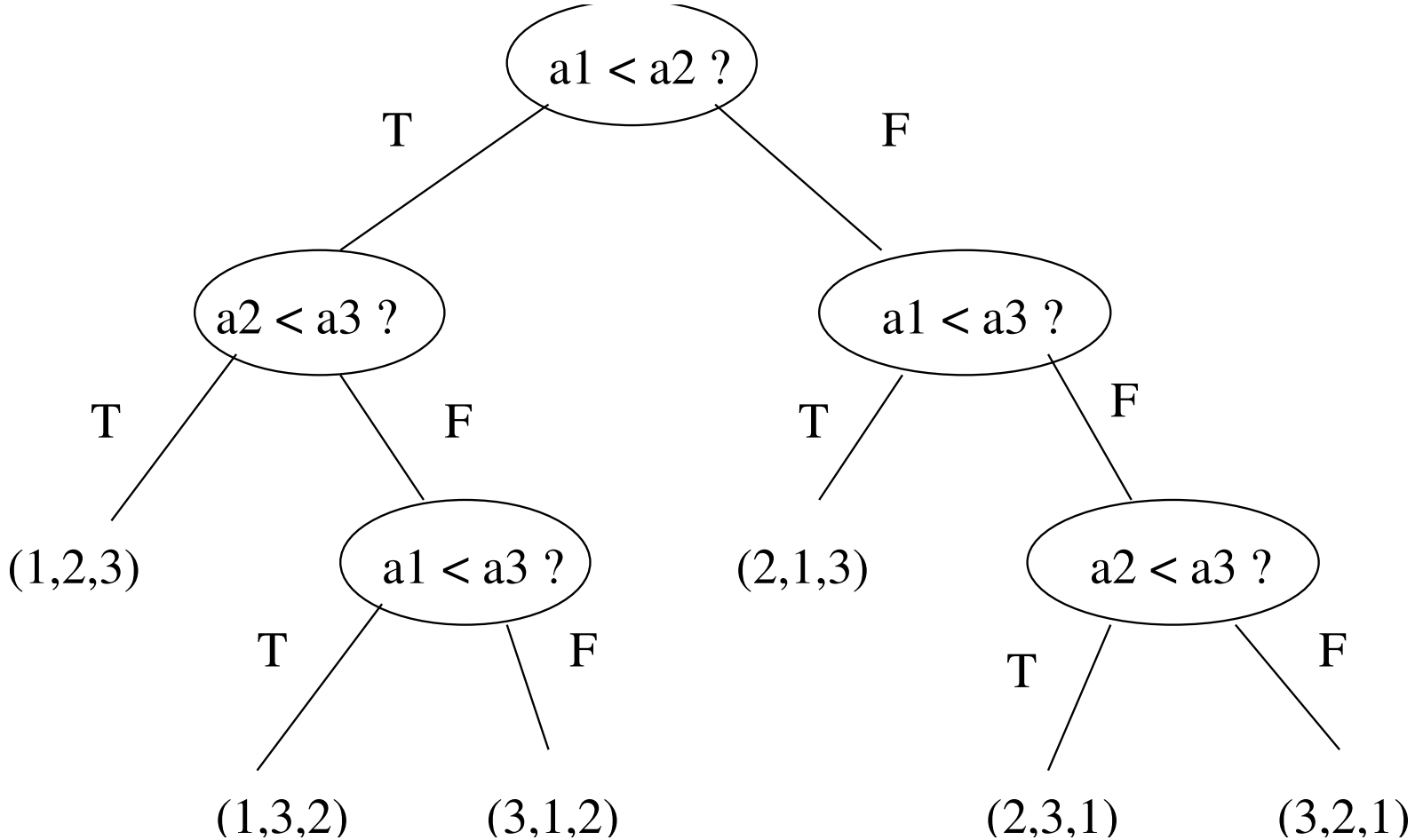
Any comparison-based sorting program can be thought of as defining a decision tree of possible executions.

Can we sort in $o(n \lg n)$?

Any comparison-based sorting program can be thought of as defining a decision tree of possible executions.
Running the same program twice on different permutations of the data causes a different sequence of comparisons to be made on each.

Decision Tree Worksheet

Decision Tree for Insertion Sort



Comparison Sorts and Decision Trees

- Since different permutations of n elements require different sequences of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree.

Comparison Sorts and Decision Trees

- Since different permutations of n elements require different sequences of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree.
- So, at least $n!$ different leaves in this tree.

Comparison Sorts and Decision Trees

- Since different permutations of n elements require different sequences of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree.
- So, at least $n!$ different leaves in this tree.
- Tree of height h has at most 2^h leaves.

Comparison Sorts and Decision Trees

- Since different permutations of n elements require different sequences of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree.
- So, at least $n!$ different leaves in this tree.
- Tree of height h has at most 2^h leaves.
- Tree must have height h such that $2^h \geq n!$

Comparison Sorts and Decision Trees

- Since different permutations of n elements require different sequences of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree.
- So, at least $n!$ different leaves in this tree.
- Tree of height h has at most 2^h leaves.
- Tree must have height h such that $2^h \geq n!$
- $h \geq \lg(n!) = \Theta(n \lg n)$

Comparison Sorts and Decision Trees

- Since different permutations of n elements require different sequences of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree.
- So, at least $n!$ different leaves in this tree.
- Tree of height h has at most 2^h leaves.
- Tree must have height h such that $2^h \geq n!$
- $h \geq \lg(n!) = \Theta(n \lg n)$
- $h =$ worst-case complexity of sorter.

Radix Sort

- Assume that the input is provided as a linked list.
- Assume that each integer has d digits.
- There will be d passes starting with the least significant digit.

Example on Board

Time Complexity

- $O(n + r)$ per pass - r is the number of bins.
- d passes, so $O(d(n + r))$ overall.
- numbers in the range [0,999] give $d = 3$ and $r = 10$.