Slides for a Course on the Analysis and Design of Algorithms

Chapter 2: The Divide-and-Conquer Strategy

Stephen J. Hegner
Department of Computing Science
Umeå University
Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

2. The Divide-and-Conquer Strategy

2.1 Mergesort

2.1.1 Description of the algorithm Given is an array $a[1..n]$ of integers. The algorithm sorts a into nondecreasing order using the following strategy.

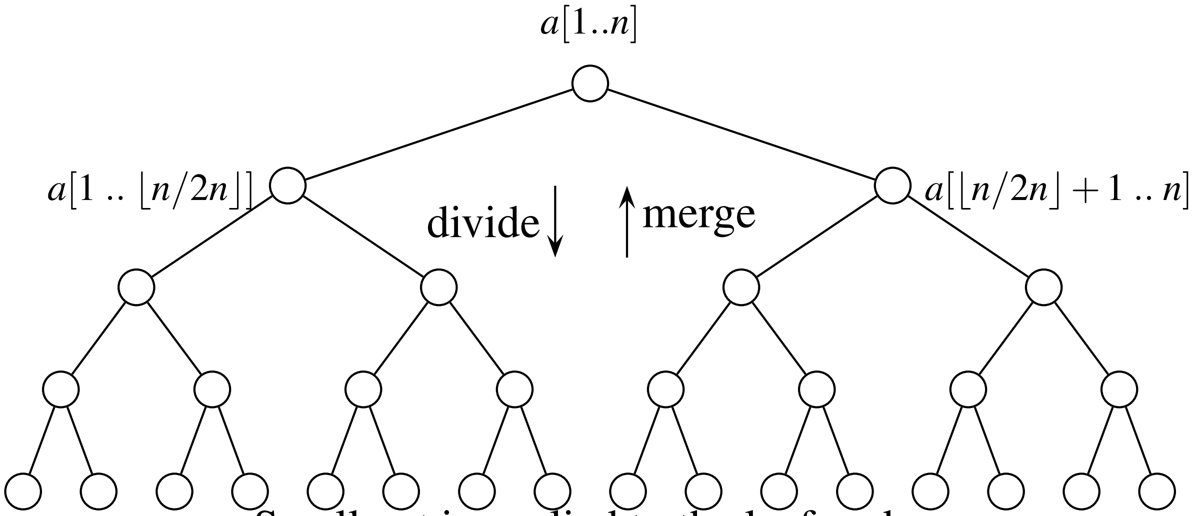
Step 1: Divide If the array is small enough, then sort it directly. Otherwise, divide it into two parts of "equal" size:

$$a\,[1\,..\,\lfloor n/2 \rfloor] \qquad a\,[\lfloor n/2 \rfloor + 1\,..\,n] \tag{1}$$

and and recursively apply the algorithm to these pieces.

Step 2: Merge the pieces Merge the two sorted pieces into one sorted list.

- $\lfloor x \rfloor$ denotes the *floor* of x; the largest integer which is no larger than x.
- Any array of size 1 is trivially sorted. Thus, if "small enough" is taken to be size 1, then no auxiliary sort program is needed.



Smallsort is applied to the leaf nodes.

```
procedure mergesort( ref a : array[1..n], low, high : 1..n)
  /* switchpt is a global constant. */
  ⟨ if (high − low ≤ switchpt)
      then
          smallsort(a, low, high);
      else
          ⟨ mid ← ⌊(low + high)/2⌋;
            mergesort(a, low, mid);
            mergesort(a, mid + 1, high);
            merge(a, low, mid, high)
          ⟩
  ⟩
procedure merge( ref a : array[1..n]; low, mid, high : [1..n])
  ⟨ b : array[low..high]; /* local array */
    p1, p2, p : integer; /* local variables */
    p1 ← low; p2 ← mid + 1; p ← low;
    while (p1 ≤ mid and p2 ≤ high) do
          ⟨ if (a[p1] ≤ a[p2])
               then ⟨ b[p] ← a[p1]; p1 ← p1 + 1; ⟩
               else ⟨ b[p] ← a[p2]; p2 ← p2 + 1; ⟩
            p ← p + 1;
          ⟩
          if (p1 ≤ mid)
            then a[p..high] ← a[p1..mid];
          a[low..p] ← b[low..p];
  ⟩

mergesort(a, 1, n); /* to sort a[1..n] */
```

2.1.2 The complexity of mergesort First, assume that $switchpt = 1$. The relevant recurrence is then

$$T(n) = \underbrace{T(\lfloor n/2 \rfloor)}_{\text{sort } a[low..mid]} + \underbrace{T(\lfloor (n+1)/2 \rfloor)}_{\text{sort } a[mid+1..high]} + \underbrace{k \cdot n}_{\text{merge}}$$

For $n = 2^m$, this becomes

$$T(2^m) = 2 \cdot T(2^{m-1}) + k \cdot 2^m$$

which by virtue of 1.10.1 has a solution of the form

$$T(n) = c_1 \cdot n + c_2 \cdot n \cdot \log(n)$$

which is in $\Theta(n \cdot \log(n))$ provided $c_2 \neq 0$. Since $T(n_1) \leq T(n_2)$ for $n_1 \leq n_2$, this complexity must hold for all n, and not just powers of two. Thus:

- The time complexity of mergesort is $\Theta(n \cdot \log(n))$.

Note further that:

- This time complexity is essentially independent of the initial order of the array. It does not matter whether the array is already sorted, in reverse order, in random order, or whatever.
- The space complexity is $\Theta(n)$ (obvious).
- If $switchpt > 1$, the asymptotic complexity remains the same. (Just substitute $n/switchpt$ for n in the analysis — Think of the input as consisting of $n/switchpt$ blocks of size $switchpt$).
- Mergesort is *stable*; that is, if the list to be sorted has duplicate keys, the relative order of the records with such keys is preserved.

2.2 Binary Search

2.2.1 Description of the algorithm

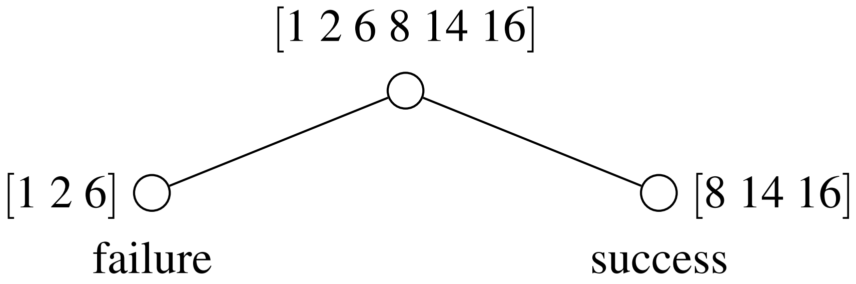
Given: a : array$[1..n]$ of integer; $/* \mathit{Sorted} */$
m : integer;
Find: $i \in [1..n]$ such that $a[i] = m$ if such an i exists,
else report failure.

The following is the "naïve" strategy, which illustrates clearly the divide-and-conquer nature of this algorithm.

```
procedure binsearch($a$ : array; $low, high$ : $[1..n]$; $m$ : integer)
  ⟨ if (($a[low] > m$) or ($a[high] < m$))
      then fail;
      else ⟨ $mid \leftarrow \lfloor (low + high)/2 \rfloor$;
             if $a[mid] = m$
               then return $mid$;
               else /* Naïve divide; one case always fails */
                  ⟨ binsearch($a, low, mid - 1, m$);
                    binsearch($a, mid + 1, high, m$); ⟩
           ⟩
  ⟩
```

2.2.2 Example

a = [1 2 6 8 14 16]
m = 14



The following is the more reasonable strategy, which “hides” the divide-and-conquer nature.

```
procedure binsearch($a$ : array; $low, high$ : $[1..n]$; $m$ : integer)
  ⟨ if (($a[low] > m$) or ($a[high] < m$))
      then fail;
      else
         ⟨ $mid \leftarrow \lfloor (low + high)/2 \rfloor$;
            case mid
               $a[mid] = m$ :  return mid;
               $a[mid] > m$ :  binsearch($a, low, mid - 1, m$);
               $a[mid] < m$ :  binsearch($a, mid + 1, high, m$);
            end case
         ⟩
      ⟩
```

Although the asymptotic complexities will be the same, the analyses are slightly different. For simplicity, it is the above version of binary search which will be analyzed.

The time complexity of binary search

For simplicity, in that which follows, it will be assumed that the list to be searched does not contain duplicates.

2.2.3 Best- and worst-case complexity

Best case: In the best case, the target is found on the first try; the complexity is thus $\Theta(1)$.

Worst case: In the worst case, the following recurrence relation holds:

$$T(n) = T(\lfloor n/2 \rfloor) + k$$

in which k is the overhead in the algorithm beyond the recursive call to *binsearch*.

Substituting 2^m for n, and writing $\hat{T}(m)$ for $T(2^m)$, the following is obtained:

$$\hat{T}(m) = \hat{T}(m-1) + k$$

The characteristic polynomial of this recurrence is

$$(x-1) \cdot (x-1)$$

and so the solutions have the form

$$\hat{T}(m) = c_1 \cdot 1^m + c_2 \cdot m \cdot 1^m = c_1 + c_2 \cdot m$$

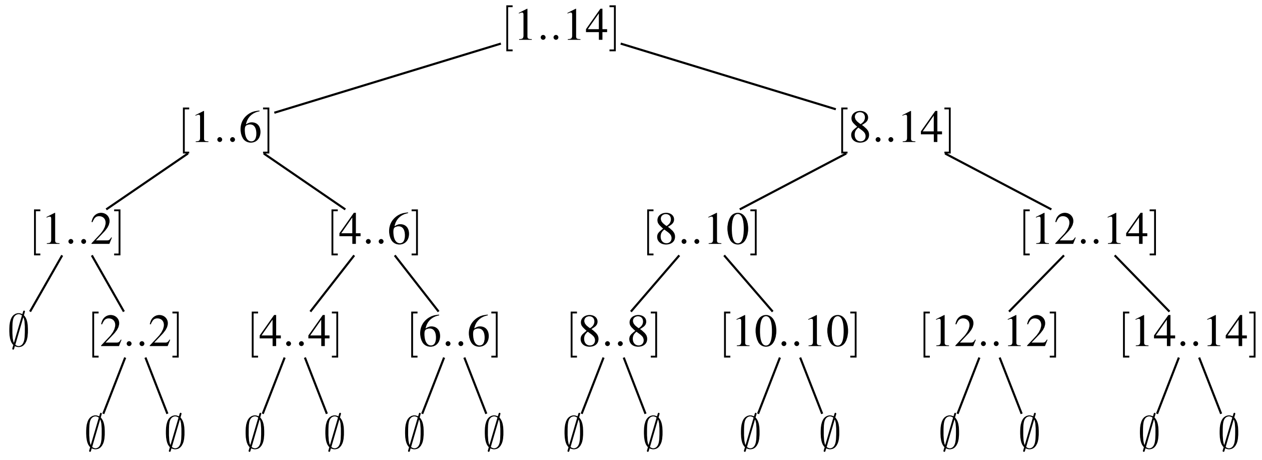
Thus,

$$T(n) = c_1 + c_2 \cdot \log(n)$$

with c_1 and c_2 constants. Thus, the worst-case time complexity is $\Theta(\log(n))$.

The analysis of the average case is somewhat more complex. To begin, the concept of a decision tree is presented.

2.2.4 Decision trees A *decision tree* represents the sequence of calls which is made for a given data item. Let $\mathbf{D}_n$ denote the decision tree for an n-element array. Shown below is $\mathbf{D}_{14}$.



Notation: Each $\emptyset$ denotes an "empty" node.
Note that:

- Each successful call terminates at an *interior* ("$[p..q]$") node. (The value found is the midpoint of $[p..q]$.)
- Each unsuccessful call terminates at an *exterior* ("$\emptyset$") node.

Thus:

- The average number of calls for a successful search = average length of a path from the root to an interior node + 1.
- The average number of calls for an unsuccessful search = average length of a path from the root to an exterior node + 1.

2.2.5 Node counts and path lengths For a given binary tree R, define the following:

- The number of *internal nodes* is denoted $\mathsf{IntNode}(R)$.
- The number of *external nodes* (or *leaf nodes*) is denoted $\mathsf{LeafNode}(R)$.
- The *internal path length*, denoted $\mathsf{IPL}(R)$, is the sum of the lengths of all paths from the root node to an interior node.
- The *external path length*, denoted $\mathsf{EPL}(R)$, is the sum of the lengths of all paths from the root node to a leaf node.

2.2.6 Example For the decision tree $\mathbf{D}_{14}$ of 2.2.4:

$$\begin{aligned}
\mathsf{IntNode}(\mathbf{D}_{14}) &= 14 \\
\mathsf{LeafNode}(\mathbf{D}_{14}) &= 15 \\
\mathsf{IPL}(\mathbf{D}_{14}) &= 1\cdot 0+2\cdot 1+4\cdot 2+7\cdot 3=31 \\
\mathsf{EPL}(\mathbf{D}_{14}) &= 1\cdot 3+14\cdot 4=59
\end{aligned}$$

2.2.7 Comment Note that

$$\mathsf{IntNode}(\mathbf{D}_n)=n$$

is always true for any $n\in\mathbb{N}$, just by definition. The fact that

$$\mathsf{LeafNode}(\mathbf{D}_n)=\mathsf{IntNode}(\mathbf{D}_n)+1$$

will be shown in 2.2.11 below. First, the relationship between $\mathsf{EPL}(\mathbf{D}_n)$ and $\mathsf{IPL}(\mathbf{D}_n)$ is developed.

2.2.8 Lemma *For any binary tree R whatever:*

$$\mathsf{EPL}(R) = \mathsf{IPL}(R) + 2 \cdot \mathsf{IntNode}(R)$$

PROOF: The proof is by induction on the size of $\mathsf{IntNode}(R)$. For simplicity of notation, let $n = \mathsf{IntNode}(R)$.

Basis: This is trivial, since for $n = 0$, $\mathsf{EPL}(R) = \mathsf{IPL}(R) = 0$.

Inductive step: Suppose that the assertion is true for a given n. Let R_{n+1} denote any binary tree with $\mathsf{IntNode}(R_{n+1}) = n + 1$. At least one of these nodes a must have two "$\emptyset$" children; *i.e.*, the subtree with it as root must be of the form $\overset{a}{\emptyset \diagup \diagdown \emptyset}$. Replace this subtree with $\emptyset$; *i.e.*, effect a transformation of the form $\overset{a}{\emptyset \diagup \diagdown \emptyset} \rightsquigarrow \emptyset$. The resulting tree $\tilde{R}_{n+1}$ has n internal nodes, so $\mathsf{EPL}(\tilde{R}_{n+1}) = \mathsf{IPL}(\tilde{R}_{n+1}) + 2 \cdot n$ in view of the induction hypothesis. However, R_{n+1} is obtained from $\tilde{R}_{n+1}$ by changing:

- one external node to an internal node; and
- adding two new external nodes.

Let k denote the path length from the root to the node labelled a. Then, in total, in transforming from $\tilde{R}_{n+1}$ to R_{n+1}:

- k has been added to $\mathsf{IPL}(\tilde{R}_{n+1})$;
- k has been subtracted from $\mathsf{EPL}(\tilde{R}_{n+1})$;
- $2 \cdot (k+1)$ has been added to $\mathsf{EPL}(\tilde{R}_{n+1})$;

i.e.,

- $\mathsf{IPL}(R_{n+1}) = \mathsf{IPL}(\tilde{R}_{n+1}) + k$; and
- $\mathsf{EPL}(R_{n+1}) = \mathsf{EPL}(\tilde{R}_{n+1}) + k + 2$;

which implies $\mathsf{EPL}(R_{n+1}) = \mathsf{IPL}(R_{n+1}) + 2 \cdot (n+1)$, as required. $\square$

2.2.9 Notation Let $\mathsf{Succ(n)}$ (resp. $\mathsf{UnSucc(n)}$) denote the average number of calls to *binsearch* in a successful (resp. unsuccessful) search of a list with n elements. In this context of searching an array $a[1..n]$ of n elements, it will always be assumed that the element m to be found has the property that $a[1] \leq m \leq a[n]$. This keeps trivial cases from corrupting the interesting cases of average time complexity.

The proof of the following is immediate.

2.2.10 Proposition *For any $n \in \mathbb{N}$,*

$$\mathsf{Succ(n)} = 1 + \mathsf{IPL}(\mathbf{D}_n)/n$$

$\square$

2.2.11 Lemma *For any nonempty binary tree R,*

$$\mathsf{LeafNode}(R) = \mathsf{IntNode}(R) + 1$$

PROOF: Let n denote the number of interior nodes.

Basis: The basis is for $n = 1$; this case is obvious.

Inductive step: Assume that the statement is true for a given $n > 1$, and let R be a binary tree with $\mathsf{IntNode}(R) = n + 1$. As argued in 2.2.8, R must have a subtree of the form $\overset{a}{\emptyset \diagup\,\diagdown \emptyset}$. Replace this tree with $\emptyset$; *i.e.*, perform a transformation of the form $\overset{a}{\emptyset \diagup\,\diagdown \emptyset} \rightsquigarrow \emptyset$, and call the resulting tree $\tilde{R}$. $\tilde{R}$ must have n internal nodes, so the inductive hypothesis may be applied to it. However, R is obtained from $\tilde{R}$ by increasing both the number of internal nodes and the number of external nodes by one, whence the result. $\square$

2.2.12 Proposition *For any $n \in \mathbb{N}$,*

$$\mathsf{UnSucc(n)} = 1 + \mathsf{EPL}(\mathbf{D}_n)/(n+1)$$

□

2.2.13 Proposition *For any $n \in \mathbb{N}$,*

$$\mathsf{Succ(n)} = (1 + 1/n) \cdot \mathsf{UnSucc(n)} - (2 + 1/n)$$

PROOF:

$$\begin{aligned} \mathsf{Succ(n)} &= 1 + \mathsf{IPL}(\mathbf{D}_n)/n \\ &= 1 + (\mathsf{EPL}(\mathbf{D}_n) - 2 \cdot n)/n \\ &= 1 + ((\mathsf{UnSucc(n)} - 1) \cdot (n+1) - 2 \cdot n)/n \\ &= (1 + 1/n) \cdot \mathsf{UnSucc(n)} - (2 + 1/n) \quad \square \end{aligned}$$

2.2.14 Theorem – average time complexity *Let* $\mathsf{a}[1..n]$ *be a sorted array containing n distinct integers, and let $m \in \mathbb{Z}$ have the property that* $\mathsf{a}[1] \le m \le \mathsf{a}[n]$. *Then the search for m has the following average time complexities.*

$$\begin{aligned} \mathsf{Succ(n)} &= \Theta(\log(n)) \\ \mathsf{UnSucc(n)} &= \Theta(\log(n)) \end{aligned}$$

PROOF: $\mathsf{UnSucc(n)}$ will always be $\Theta(\log(n))$, since an unsuccessful search will always use $\log_2(n)$ or $\log_2(n+1)$ calls to reach a leaf node. The complexity of $\mathsf{Succ(n)}$ then follows from 2.2.13. □

2.3 Quicksort

2.3.1 Informal comparison of mergesort and quicksort To sort $a[low..high]$:

Mergesort:

1. Divide a into $a[low..\alpha]$, $a[\alpha+1..high]$.
2. Sort the two pieces separately.
3. Merge the two sorted pieces into one.

Quicksort:

1. Rearrange a so that each element of $a[low..\alpha]$ is smaller than each element of $a[\alpha+1..high]$.
2. Sort $a[low..\alpha]$, $a[\alpha+1..high]$ separately.
3. Note that no merging is necessary.

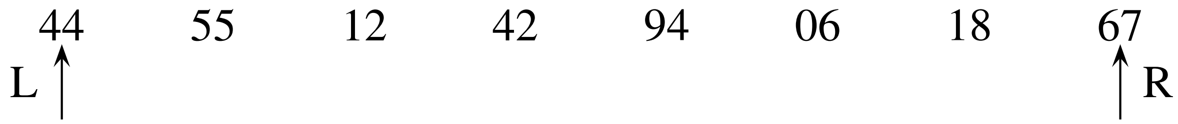
2.3.2 Example of quicksort

Let

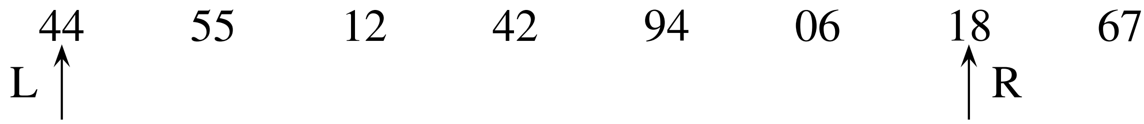
$$a = 44 \quad 55 \quad 12 \quad 42 \quad 94 \quad 06 \quad 18 \quad 67$$

The first step is to pick a “partition” point; this value must lie between the least and greatest element. In this example, choose 43.

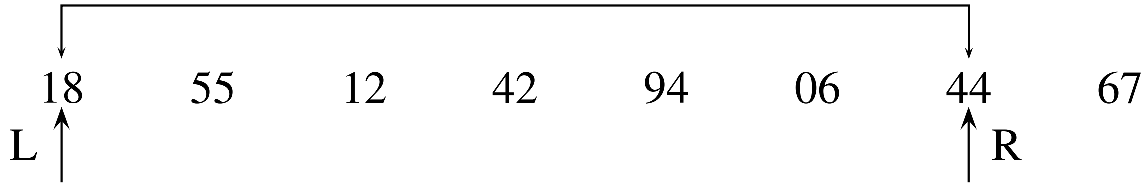
- Begin with pointers L and R which point to the leftmost and rightmost elements, respectively:



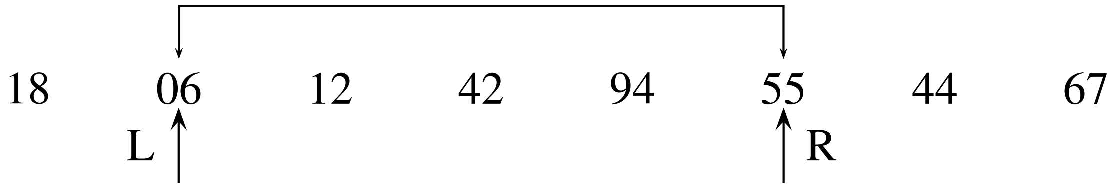
- Move pointer L to the right until finding an element $>$ 43, and move pointer R to the left until finding an element $<$ 43:

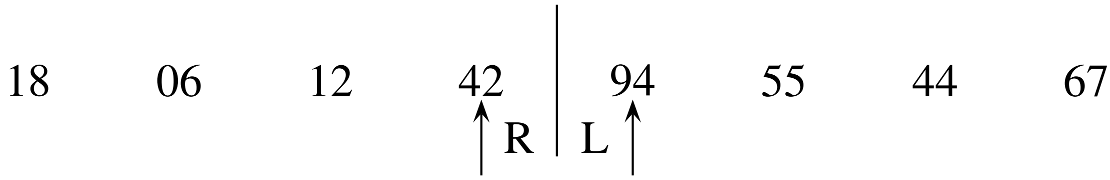


- If L and R have not met, swap these two elements:



- Repeat this process until pointers L and R cross over each other or meet:





- The process now continues by sorting each of the two blocks separately.
- Note that no merging will be necessary, since every element in the left block is smaller than every element in the right block.

2.3.3 Possible strategies for pivot selection

- The key step is the selection of the partition (or *pivot*) element.
- A strategy is sought which yields a division of the array into two part of approximately equal size.
- Some possible strategies are the following:
  1. Select a value at random from amongst the possible key values.
     - It is often a good choice.
     - The problem is that it may yield a value which is either larger or smaller than all keys.
  2. Select a random array element.
     - It is often a good choice.
     - The problem with a random value is avoided.
     - An extreme value might be selected.
  3. Select the leftmost element.
     - It is often a good choice if the array is random.
     - It is a very bad strategy is the array is even partially sorted.
  4. Compute the average of a few elements.
     - This strategy is usually better than the above, and only slightly slower.
  5. Compute the average of all elements.
     - This is a good strategy, but quite slow.
  6. Compute the median of all elements.
     - This will yield an optimal pivot element, but is too slow to use in practice.

2.3.4 The general partition algorithm

The following algorithm assumes only that the pivot element is within the range of the elements to be partitioned.

- This pivot element is computed by the procedure *getpivot.*
- The parameter *divider* identifies the position of the rightmost element of the left interval in the partition.
- The array must contain at least two elements; *i.e.*, $low < high$.

```
procedure partition( ref a : array [1..n] of keytype;
                     low, high : 1..n;
                     ref divider : 1..n);
  ⟨ ref left, right : 0 .. n + 1;
    part : keytype;
    $left \leftarrow low - 1$; $right \leftarrow high + 1$; $part \leftarrow getpivot(a, low, high)$;
    while ($left < right$) do
        ⟨ /* BlockA */
          $left \leftarrow left + 1$; $right \leftarrow right - 1$;
          while ($a[left] < part$) do $left \leftarrow left + 1$;
          while ($a[right]) > part$) do $right \leftarrow right - 1$;
          if ($left < right$) then $swap(a[left], a[right])$;
        ⟩
        if ($left = right = high$) then $right \leftarrow right - 1$;
        $divider \leftarrow right$;
  ⟩
```

2.3.5 Definition To establish the correctness of this partitioning algorithm, some predicates are necessary.

(a) Let LegalPiv denote the predicate which asserts that

$$\min(\{a[i] \mid low \le i \le high\}) \le part \le \max(\{a[i] \mid low \le i \le high\})$$

(b) Let SwapBd denote the predicate which asserts that

$$\begin{gathered}(\forall x)((x \in \{a[i] \mid low \le i < left\}) \Rightarrow (x \le part)) \;\wedge \\ (\forall x)((x \in \{a[i] \mid right < i \le high\}) \Rightarrow (part \le x))\end{gathered}$$

(c) A predicate α is called an *invariant* of the program block B if, whenever α is true at the start of B, it is also true upon completion of B.

2.3.6 Lemma *Assume that predicate* LegalPiv *is true upon entry to block A of the procedure of 2.3.4. Then* SwapBd *is an invariant of block A.*

PROOF: First of all, observe that condition LegalPiv, together with the fact that the movement of *left* and *right* ceases as soon as they meet or cross, ensures that the indices *left* and *right* can never go "out of bounds"; that is, it is always the case that $left \le high$ and $right \ge low$.

Next, suppose that SwapBd is true at the beginning of an execution of block A. Then, it is clearly true after the execution of the two while loops. The "swap" block also maintains SwapBd, because the condition SwapBd only specifies properties for elements strictly to the left of *left* and strictly to the right of *right*, and these values are not changed by the swap. □

2.3.7 Theorem *If* LegalPiv *is true upon entry to block A, then* SwapBd *is an invariant of the entire while loop containing A.* □

2.3.8 Definition Let Divider denote the predicate

$$\begin{aligned} (\forall x)((x \in \{a[i] \mid \mathit{low} \le i \le \mathit{divider}\}) \Rightarrow (x \le \mathit{part})) \ \wedge \\ (\forall x)((x \in \{a[i] \mid \mathit{divider}+1 \le i \le \mathit{high}\}) \Rightarrow (\mathit{part} \le x)) \ \wedge \\ (\mathit{low} \le \mathit{divider} < \mathit{high}) \end{aligned}$$

2.3.9 Theorem – partition works correctly *If getpivot provides a value of part for which* LegalPiv *is true, then* Divider *is true upon completion of procedure partition.*

PROOF: In view of 2.3.7, it suffices to check that $\mathit{divider} \leftarrow \mathit{right}$ assigns the correct value to divider.

- If $\mathit{left} > \mathit{right}$ holds at lines 14-15 of 2.3.4, it must be the case that $a[x] = \mathit{part}$ for all x in the range $\mathit{left}+1 \le x \le \mathit{right}-1$, so by the invariance of SwapBd established in 2.3.6, it must be the case that the choice of $\mathit{divider} = \mathit{right}$ is correct.
- If $\mathit{left} = \mathit{right}$, then it must be the case that $a[\mathit{left}] = a[\mathit{right}] = \mathit{part}$.
  - ➢ If $\mathit{low} < \mathit{left} = \mathit{right} < \mathit{high}$, then either $\mathit{divider} = \mathit{right}$ or $\mathit{divider} = \mathit{right} - 1$ will work.
  - ➢ If $\mathit{low} = \mathit{left} = \mathit{right}$, then only $\mathit{divider} = \mathit{right}$ will work.
  - ➢ If $\mathit{left} = \mathit{right} = \mathit{high}$, then only $\mathit{divider} = \mathit{right} - 1$ will work, since $\mathit{divider} = \mathit{high}$ would result in no elements to the right of *divider*. Thus, *right* must be decreased by one.

□

2.3.10 The full quicksort algorithm

The full quicksort algorithm just calls partition recursively.

```
procedure quicksort( ref a :  array[1..n] of  integer;
                              low, high : 1..n);
   ⟨
      divider : 1..n;
      if (low < high) then ⟨ partition(a, low, high, divider);
                               quicksort(a, low, divider);
                               quicksort(a, divider + 1, high);
                             ⟩
   ⟩
```

The starting point is just

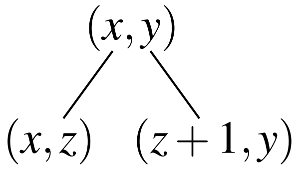
$quicksort(a, 1, n);$

The time complexity of quicksort

2.3.11 Call trees Given a particular instance I of the array $a[1..n]$ and a fixed choice for the function *getpivot*, the *call tree* $\mathsf{CallTree}(I, \mathit{getpivot})$ is the binary tree which shows the recursive nesting of calls to quicksort. Formally,

(a) The root is labelled $(1,n)$.

(b) The node labelled (x,y), with $x < y$, has children as shown below



(x,y)

(x,z) $(z+1,y)$

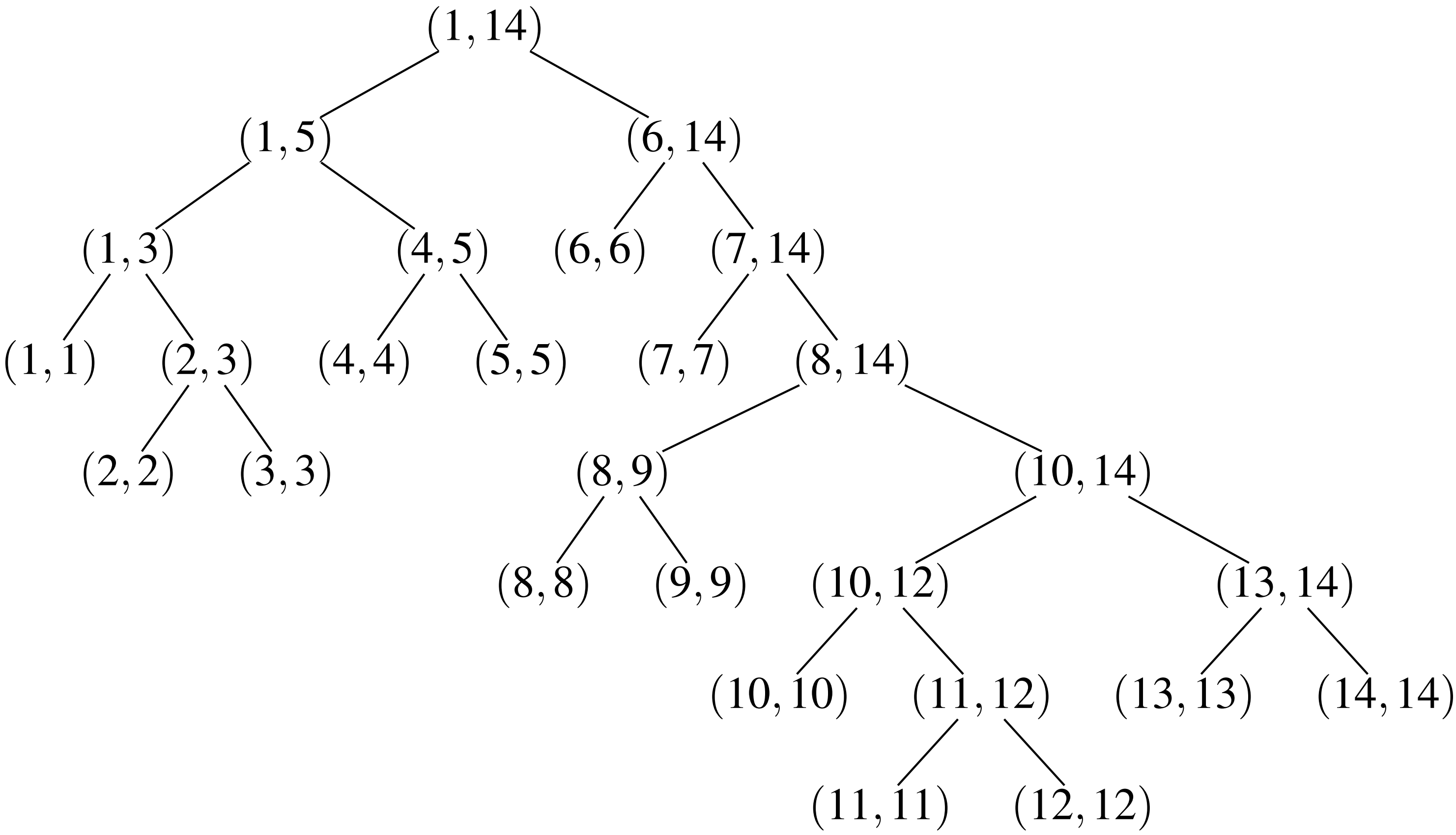
with z the value for *divider* returned by the call

$$\mathit{partition}(a, x, y, \mathit{divider})$$

(c) A node labelled (x,x) for any x has no descendants.

(d) If v is a node in such a tree, and the label for v is (x,y), then the notation $x = \mathsf{Low}(v)$, $y = \mathsf{High}(v)$ will be used, so that $(x,y) = (\mathsf{Low}(v), \mathsf{High}(v))$.

2.3.12 Example Shown below is a possible call tree for a fourteen-element array.



2.3.13 Conventions used in complexity analysis

- In a call tree such as shown above in 2.3.12, nodes of the form (x, x) will be taken to be leaf nodes. These nodes have no $\emptyset$ descendants, as is the case with the decision trees of 2.2.4. This convention is critical to the definition of external path length, as given in 2.2.5.

- It will always be assumed that the pivot selection routine $getpivot(a, low, high)$ runs in time $O(high - low)$. This condition is met by all alternatives mentioned in 2.3.3.

2.3.14 Lemma *The time complexity for a call* $partition(a, low, high, divider))$ *is* $\Theta(high - low)$ *in all cases.* □

2.3.15 Notation Let R be any binary tree whatever. For any node v of R, let $\mathsf{Depth}(v)$ denote the length of the path from the root node to v. Note that the root node has depth 0 (and not 1) under this definition.

2.3.16 Lemma *Let I be an instance for the array* $\mathsf{a}[1..n]$ *of integers, and let* $\mathsf{CallTree}(I, getpivot)$ *be the corresponding call tree for pivot function getpivot. Then*

$$\sum_{v \in \mathsf{Vertices}(\mathsf{CallTree}(I, getpivot))} (\mathsf{High}(v) - \mathsf{Low}(v) + 1) \quad = \quad \mathsf{EPL}(\mathsf{CallTree}(I, getpivot)) + n$$

PROOF:

$$\begin{aligned}
\sum_{v \in \mathsf{Vertices}(\mathsf{CallTree}(I, getpivot))} (\mathsf{High}(v) - \mathsf{Low}(v) + 1) &= \sum_{i=1}^{n} \#(i) \\
&= \sum_{i=1}^{n} (\mathsf{Depth}((i,i)) + 1) \\
&= \sum_{i=1}^{n} \mathsf{Depth}((i,i)) + n \\
&= \mathsf{EPL}(\mathsf{CallTree}(I, getpivot)) + n
\end{aligned}$$

In the first line $\#(k)$ denotes the number of times that the index k occurs in a node labelled (x,y), in the sense that $x \leq k \leq y$. It is easy to see that the sum $\sum(\mathsf{High}(v) - \mathsf{Low}(v) + 1)$ counts exactly such occurrences over all i; there are exactly $y - x + 1$ such occurrence in the node labelled (x,y), whence the first equality. Next, the index k occurs exactly in those nodes which lie along the path from the root to the node labelled (k,k); there are $\mathsf{Depth}(k) + 1$ such nodes. This establishes the second equality. The final two are trivial. □

2.3.17 Theorem *Let* $\mathsf{a}[1..n]$ *be an array of integers, and let I be an instance of values for this array. The running time for quicksort on I is* $\Theta(\mathsf{EPL}(\mathsf{CallTree}(I, getpivot)))$, *with* $\mathsf{CallTree}(I, getpivot)$ *the particular call tree which the function partition yields.*

PROOF: The proof follows immediately from 2.3.14 and 2.3.16. Note that $\Theta(high - low) = \Theta(high - low + 1)$ and that $\Theta(\mathsf{EPL}(\mathsf{CallTree}(I, getpivot))) = \Theta(\mathsf{EPL}(\mathsf{CallTree}(I, getpivot)) + n)$, the latter since $\mathsf{EPL}(\mathsf{CallTree}(I, getpivot)) > n$. □□

2.3.18 Definition – almost balanced Let R be any binary tree whatever. Call R *almost balanced* if for any two leaf nodes v_1 and v_2 of R, $|\mathsf{Depth}(v_1) - \mathsf{Depth}(v_2)| \leq 1$.

2.3.19 Corollary *Let* $\mathsf{a}[1..n]$ *be an array of integers, and let I be an instance of values for this array.*

(a) *The best-case running time for quicksort is* $\Theta(n \cdot \log(n))$.

(b) *The worst-case running time for quicksort is* $\Theta(n^2)$.

PROOF: The best case occurs when $\mathsf{CallTree}(I, getpivot)$ is almost balanced. It is easy to see that there is always a partitioning which yields such a tree. In that case, there are n leaves, each with a depth of approximately $\log(n)$, for a total external path length in $\Theta(n \cdot \log(n))$.

The worst case occurs when each partition of an interval (x, y) yields intervals (x, x) and $(x+1, y)$. In that case, the external path length is $\sum_{i=1}^{n} i = n \cdot (n+1)/2 \in \Theta(n^2)$. □

Next, the question of average time complexity for quicksort is examined.

2.3.20 Conventions In the analysis of the time complexity of quicksort in the average case, the following assumptions are made:

- All values in the array $a[1..n]$ are distinct.
- All configurations I are equally likely.
- The pivot element is chosen at random from amongst the values stored in $a[1..n]$.

2.3.21 The recurrence in the average case Let $T_A(n)$ denote the average number of comparisons required to sort an n-element list with quicksort. The following inequality then holds for $n > 1$:

$$\begin{aligned} T_A(n) &\leq k_1 \cdot n + \frac{1}{n-1} \cdot \left(\sum_{i=1}^{n-1} (T_A(i) + T_A(n-i)) \right) \\ &= k_1 \cdot n + \frac{2}{n-1} \cdot \sum_{i=1}^{n-1} T_A(i) \end{aligned}$$

In the first line, the $k_1 \cdot n$ term represents the amount of time required to partition the array. The second term represents the amount of time need to sort recursively each component of the partition, averaged over all possibilities. The $n-1$ represents the number of distinct sizes for the two pieces of the partition; if the sizes are i and $n-i$ respectively, then the time to sort recursively these pieces is $T_A(i) + T_A(n-i)$. For $n = 1$, the time required is just some contant, which may be taken to be k_1. More precisely, k_1 may be chosen to be large enough to satisfy both conditions.

2.3.22 Lemma *For the average case of quicksort,*

- $T_A(1) \leq k_1$.
- $T_A(2) \leq 4 \cdot k_1$.
- *For* $n \geq 3$, $T_A(n) \leq 4 \cdot k_1 \cdot n \cdot \log_2(n-1)$.

PROOF:

Basis: The cases $n = 1$ and $n = 2$ are trivial.

Inductive step: Fix $n \geq 2$, and assume that the statement is true for all k with $k \leq n$. Then, the argument on the next slide shows that

$$T_A(n+1) \leq 4 \cdot n \cdot \log_2(n)$$

□

2.3.23 Theorem *Quicksort has time complexity* $\Theta(n \cdot \log(n))$ *in the average case, with n the size of the input array.*

PROOF: It is $O(n \cdot \log(n))$ in view of the above lemma, but since the best case is $\Theta(n \cdot \log(n)))$, the average case can be no better. □

Grinding the math for 2.3.22:

$$
\begin{aligned}
& T_A(n+1) \\
\leq\ & k_1 \cdot (n+1) + \frac{2}{n} \cdot \sum_{i=1}^{n} (T_A(i)) \\
=\ & k_1 \cdot (n+1) + \frac{2}{n} \cdot \left(T_A(1) + T_A(2) + \sum_{i=3}^{n} T_A(i) \right) \\
=\ & k_1 \cdot (n+1) + \frac{2}{n} \cdot \left(T_A(1) + T_A(2) + \sum_{i=3}^{n} (4 \cdot k_1 \cdot (i-1) \cdot \log_2(i-1)) \right) \\
=\ & k_1 \cdot \left(n+1 + \frac{10}{n} + \frac{8}{n} \cdot \sum_{i=2}^{n-1} i \cdot log_2(i) \right) \\
\leq\ & k_1 \cdot \left(n+1 + \frac{10}{n} + \frac{8}{n} \cdot \int_2^n i \cdot log_2(i) \cdot di \right) \\
=\ & k_1 \cdot \left(n+1 + \frac{10}{n} + \frac{8}{n \cdot \log_e(2)} \cdot \int_2^n i \cdot log_e(i) \cdot di \right) \\
=\ & k_1 \cdot \left(n+1 + \frac{10}{n} + \frac{8}{n \cdot \log_e(2)} \cdot \left(\frac{i^2 \cdot log_e(i)}{2} - \frac{i^2}{4} \right) \Big|_2^n \right) \\
=\ & k_1 \cdot \left(n+1 + \frac{10}{n} + \frac{8}{n \cdot \log_e(2)} \cdot \left(\frac{n^2 \cdot log_e(n)}{2} - \frac{n^2}{4} - \frac{2^2 \cdot \log_e(2)}{2} + \frac{2^2}{4} \right) \right) \\
=\ & k_1 \cdot \left(n+1 + \frac{10}{n} + 4 \cdot n \cdot \log_2(n) - \frac{2 \cdot n}{\log_e(2)} - \frac{16}{n} - \frac{8}{n \cdot \log_e(2)} \right) \\
=\ & k_1 \cdot \left(4 \cdot n \cdot \log_2(n) + \left(n + 1 - \frac{6}{n} - \frac{2 \cdot n}{\log_e(2)} + \frac{8}{n \cdot log_e(2)} \right) \right) \\
=\ & k_1 \cdot \left(4 \cdot n \cdot \log_2(n) + \underbrace{\left(n \cdot \left(1 - \frac{2 \cdot n}{\log_e(2)} \right) - \frac{6}{n} + \frac{8}{n \cdot log_e(2)} \right)}_{<0 \text{ for } n \geq 3} \right) \\
\leq\ & 4 \cdot k_1 \cdot \log_2(n)
\end{aligned}
$$

2.3.24 Some final observations regarding quicksort

- In practice, quicksort appears to be two to three times faster than mergesort.
- Mergesort has the advantage that the time required to sort is relatively independent of the initial arrangement of the list. This is not the case with quicksort.
- Mergersort has the further advantage that it is stable (see 2.1.2), while quicksort is not.
- The average time complexity is still $\Theta(n \cdot \log(n))$ with duplicates in the list, but the proof is more complex.
- As is the case with mergesort, the performance may be improved by using a simpler sort for small lists.
- The space complexity is $\Theta(n)$ in all cases.

2.3.25 A simplified version of quicksort

- In the case that the pivot element is chosen as a value from the array,the quicksort algorithm can be simplified somewhat, as shown on the next slide.
- The procedure *partition1* is given a fifth argument which identifies the index in the array *a* of the pivot value.
- It then divides $a[low..high]$ into three pieces.
    - ➢ For $x < divider$, $a[x] \leq a[pivindex]$.
    - ➢ For $x > divider$, $a[x] \geq a[pivindex]$.
    - ➢ $a[divider] = a[pivindex]$.
- The recursive sort ignores $a[divider]$, since it is already in the correct position.
- Note that $divider = low$ and $divider = high$ are possible.
- The asymptotic complexities of this algorithm are the same as for the previous one, and will not be analyzed separately.
- *choose_pivot* is the pivot-selection algorithm, which returns an index in $[low..high]$.

```
procedure partition1( ref a : array [1..n] of keytype;
                       low, high : 1..n;
                       ref divider : 1..n;
                      pivindex : 1..n);
  ⟨ ref left, right : 0 .. n + 1;
    part : keytype;
    part ← a[low + pivindex − 1];
    /* Temporarily store part in the leftmost position: */
    swap(a[low], a[low + pivindex − 1]); low ← low + 1;
    left ← low − 1; right ← high + 1;
    while (left < right) do
          ⟨ /* BlockA */
            left ← left + 1; right ← right − 1;
            while (a[left] < part) do left ← left + 1;
            while (a[right] < part) do right ← right − 1;
            if (left < right) then swap(a[left], a[right]);
          ⟩
          swap(a[low], a[right]); /* Restore part to the correct position */
          divider ← right;
    ⟩

  procedure quicksort1( ref a : array[1..n] of integer;
                         low, high : 1..n);
    ⟨ divider, pivindex : 1..n;
      if (low < high) then ⟨ pivindex ← choose_pivot(a);
                              partition1(a, low, high, divider, pivindex);
                              quicksort1(a, low, divider − 1);
                              quicksort1(a, divider + 1, high);
                            ⟩
    ⟩
```

2.4 The General Divide-and-Conquer Strategy

2.4.1 The pseudocode The general divide-and-conquer strategy has the following form.

```
procedure $DC(inobj : object,\ outobj : object)$
  ⟨ $solved$ : ref boolean;
    $in1, in2, out1, out2 : object$;
    $presolve(inobj, outobj, solved)$;
    if ( not $solved$) then ⟨ $divide(inobj, in1, in2)$;
                              $DC(in1, out1)$; $DC(in2, out2)$;
                              $combine(out1, out2, outobj)$; ⟩
  ⟩
```

presolve: checks to see if the problem is simple and can be solved directly.

- Sort a small list for mergesort, quicksort.
- Failed or found in binary search.

divide: takes a single problem instance and splits it into two subinstances.

- trivial in mergesort; partition in quicksort.
- divide interval in binary search.

combine: combines the results of two solutions into one.

- merge in mergesort; trivial in quicksort.
- trivial in binary search.

2.5 Order Statistics

2.5.1 Problem description The problem of *order statistics* may be described as follows:

Given:
- array $a[1..n]$ of integer;
- $k \in 1..n$.

Find: The k^{th}-smallest element a.

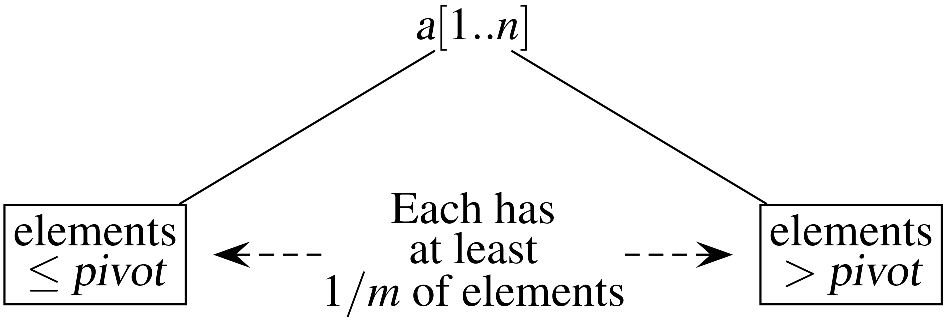
- A simple-minded solution is to sort a and then pick the k^{th} element.
- This does much more work than in actually necessary.
- A better solution is to proceed as in quicksort, but only continue to sort the "useful" half of the interval at each step.
- It is easy to see that this results in the following time complexities:

  Best case: $\Theta(n)$.
  Worst case: $\Theta(n^2)$.
  Average case: $\Theta(n)$.

- The proof is very similar to that for quicksort.
- It is, however, possible to design a divide-and-conquer algorithm which runs in time $\Theta(n)$ in the *worst* case. This algorithm is now developed.

2.5.2 The idea of a $\Theta(n)$ worst-case algorithm for order statistics

- The general idea is to proceed as in quicksort, but sorting only the "necessary" of the two partitions.
- The trick is to select the partition element in such a way that the degenerate case leading to $\Theta(n^2)$ complexity cannot occur.
- The pivot element is computed in such a way that a minimal percentage of the elements, $1/m$, lies in each of the partitions.



- The maximum depth of the tree is then $\lceil \log_{m/(m-1)}(n) \rceil$.
- If the time at each level is bounded by Θ(list length), then with $r = \frac{m-1}{m}$ and $B = \lceil \log_{m/(m-1)}(n) \rceil$, the total running time will be:

$$
\begin{aligned}
T(n) &= n \cdot \sum_{i=0}^{B} r^i \\
&= n \cdot \frac{1 - r^{B+1}}{1-r} \leq n \cdot \frac{1}{1-r} = n \cdot \text{constant}
\end{aligned}
$$

2.5.3 The median-of-medians rule

The median-of-medians rule on the array $a[1..n]$ proceeds as follows:

1. Select r, $1 \leq r \leq n$, with r odd. (Selection process discussed later.)
2. Build $\lfloor n/r \rfloor$ groups of r elements each.
3. Discard the last $n - \lfloor n/r \rfloor \cdot r$ elements.
4. For $1 \leq i \leq r$, Set $m_i \leftarrow$ median of i^{th} group.
5. Set $mm \leftarrow$ median$(\{m_i \mid 1 \leq i \leq r\})$.

2.5.4 Facts

(a) *At least* $\lceil \lfloor n/r \rfloor /2 \rceil$ *of the* m_i*'s are* $\leq mm$.

(b) *At least* $\lceil \lfloor n/r \rfloor /2 \rceil$ *of the* m_i*'s are* $\geq mm$.

(c) *At least* $\lceil r/2 \rceil \cdot \lceil \lfloor n/r \rfloor /2 \rceil$ *of the elements of* a *are* $\leq mm$.

(d) *At least* $\lceil r/2 \rceil \cdot \lceil \lfloor n/r \rfloor /2 \rceil$ *of the elements of* a *are* $\geq mm$.

(e) *At most* $n - \lceil r/2 \rceil \cdot \lceil \lfloor n/r \rfloor /2 \rceil$ *of the elements of* a *are* $> mm$.

(f) *At most* $n - \lceil r/2 \rceil \cdot \lceil \lfloor n/r \rfloor /2 \rceil$ *of the elements of* a *are* $< mm$.

PROOF: $\lceil \lfloor n/r \rfloor /2 \rceil$ is half the number of groups, rounded up. From this (a) and (b) follow immediately. (c) and (d) then follow from (a) and (b), as do (e) and (f). □

2.5.5 The high-level order-statistics algorithm The algorithm is shown on the next page. The key points are as follows.

- The algorithm calls itself in two different ways for distinct purposes.
  - ➢ to compute the median of medians, by recursively finding the median of a list (*i.e.*, by finding the $i/2$nd element in an i-element list;
  - ➢ to mimic the relevant half of quicksort, using the median of medians to define the dividing point.
- For this algorithm, the procedure *partition1* of 2.3.25 is used.

```
/*   Return the $k^{th}$ largest element in $a[low..high]$   */
function $orderstat(a:$ ref array$[1..n]$ of integer;
                    $low, high, k : 1..n)$ : integer;
  ⟨ $s, mm$ : integer;
      $r$ integer constant;
      $med$ : array$[1..\lfloor(high-low+1)/r\rfloor]$ of integer;
      $s \leftarrow high - low + 1$;
      if $(\lfloor s/r \rfloor \leq 1)$
        then $mm \leftarrow median(a[low, high])$;
        else ⟨
              for $i \leftarrow 1$ to $\lfloor s/r \rfloor$ do
                  $med[i] \leftarrow median(a[low+(i-1)\cdot\lfloor r\rfloor$
                                       $..low+(i-1)\cdot\lfloor r\rfloor+(r-1)])$;
              $mm \leftarrow orderstat(med, 1, \lfloor s/r\rfloor, \lceil\lfloor s/r\rfloor/2\rceil)$;
            ⟩
      $partition1(a, low, high, divider, mm)$;
      /* $mm$ = index to pivot value to be used */
      case
        $divider = k$ : return $a[divider]$;
        $divider > k$ : return $orderstat(a, low, divider-1, k)$;
        $divider < k$ : return $orderstat(a, divider+1, high,$
                                          $k - divider)$;
      end case
  ⟩
```

2.5.6 The recurrence defining the time complexity

- Let $T(m)$ denote the worst-case time complexity for a call to *orderstat* with $\mathit{high} - \mathit{low} + 1 = m$. A line-by-line analysis of the complexity follows. Each of the k_i is a constant.

  Line 9: k_0.

  Lines 11-13: $k_1 \cdot \lfloor m/r \rfloor \leq k_1 \cdot m$

  Line 14: $T(\lfloor m/r \rfloor)$.

  Line 16: $k_2 \cdot m$.

  Lines 18-23: $T(\text{max. no. elements in the larger part of the partition}) \leq T(m - \lceil r/2 \rceil \cdot \lceil \lfloor m/r \rfloor /2 \rceil)$. (Follows from 2.5.4 (e) and (f).)

- Thus the recurrence relation to be solved is:

$$\begin{aligned} T(m) &\leq k_0 + (k_1 + k_2) \cdot m + T(\lfloor m/r \rfloor) + T(m - \lceil r/2 \rceil \cdot \lceil \lfloor m/r \rfloor /2 \rceil) \\ &\leq k \cdot m + T(\lfloor m/r \rfloor) + T(m - \lceil r/2 \rceil \cdot \lceil \lfloor m/r \rfloor /2 \rceil) \end{aligned}$$

- It is assumed that $m \geq 1$, and so k_0 may be eliminated by choosing the constant k large enough so that $k_0 + (k_1 + k_2) \cdot m \leq k \cdot m$ for all $m \geq 1$.

- In general this is a difficult recurrence to solve. The trick is to find a value for r which works.

2.5.7 Theorem *Let* $\mathrm{a}[1..n]$ *be an* n*-element array of distinct integers. With* $r = 5$*, the worst-case time complexity of a call of the form* $\mathrm{orderstat}(\mathrm{a}, 1, n)$ *of the order-statistics program of 2.5.5 is* $\Theta(n)$.

PROOF:

$$\lceil r/2 \rceil \cdot \lceil \lfloor m/5 \rfloor /2 \rceil = 3 \cdot \lceil \lfloor m/5 \rfloor /2 \rceil \geq 3 \cdot \lfloor m/5 \rfloor /2 = 1.5 \cdot \lfloor m/5 \rfloor$$

Thus, in view of 2.5.4 (e) and (f), at most

$$m - 1.5 \cdot \lfloor m/5 \rfloor \leq m - 1.5 \cdot (m/5 - 1) \leq 0.7 \cdot m + 1.5$$

elements of $\mathrm{a}[low..high]$ will be $> mm$ (resp. $< mm$).

For $m \geq 50$, $0.7 \cdot m + 1.5 \leq 3 \cdot m/4 - 1$, so for $m \geq 50$,

$$T(m) \leq k \cdot m + T(\lfloor m/5 \rfloor) + T(\lfloor 3 \cdot m/4 \rfloor)$$

It is possible to select k large enough that

$$T(m) \leq k \cdot m \quad \text{for } m \leq 50$$

It then follows by induction that

$$T(m) \leq 20 \cdot k \cdot m$$

for all $m \geq 1$. The basis step is obvious. For the inductive step, assume that $T(p) \leq 20 \cdot k \cdot p$ for all $p < m$. Then

$$\begin{aligned} T(m) &\leq k \cdot m + T(\lfloor m/5 \rfloor) + T(\lfloor 3 \cdot m/4 \rfloor) \\ &\leq k \cdot m + \frac{1}{5} \cdot 20 \cdot k \cdot m + \frac{3}{4} \cdot 20 \cdot k \cdot m \\ &= 20 \cdot k \cdot m \end{aligned}$$

□

2.5.8 Arrays with duplicate values

- If the array $a[1..n]$ contains duplicate values, the choice of $r = 5$ may not work.
- For example, suppose that $0.7 \cdot m + 1.5$ elements are $\leq mm$, with the rest equal to mm.
- Let $T_e(m)$ denote the time for a call of the form $orderstat(a, low, divider - 1, k)$. Then

$$T_e(m) \leq T(0.7 \cdot m + 1.5 + \underbrace{\frac{1}{2} \cdot (0.3 \cdot m - 1.5)}_{\substack{\text{Assume that half} \\ \text{of the rest of the elements} \\ \text{fall into the left partition.}}}) = T(0.85 \cdot m + 0.75)$$

- A similar result hold for a call of the form $orderstat(a, divider + 1, high, k)$.
- Thus, in the worst case, the following recurrence, which is super-linear, holds:

$$T(m) \leq k \cdot m + T(\lfloor m/5 \rfloor) + T(0.85 \cdot m)$$

- Since this is an inequality, it does not prove that the algorithm is not $\Theta(n)$, but it does not substantiate that it is either.

- There are two ways to deal with this problem:
  1. Choose a different value for r.
     - The value $r = 9$ works, but the constant multipliers are larger, which results in a slower algorithm.
  2. Modify the partition algorithm to divide the set of values into three parts:

$= divider$ $< divider$ $> divider$

  - This modified partitioning may still be performed in linear time.

- Consult the text by Horowitz, Sahni, and Rajasekaran for details on how to implement these ideas.

The bottom line is the following.

2.5.9 Theorem *There is a* $\Theta(n)$ *worst-case time algorithm for the order-statistics problem, even in the case that the array contains duplicate elements.* □

2.5.10 Improving the worst-case time complexity of quicksort

- If *getpivot* in the program of 2.3.4 is implemented using a $\Theta(n)$ order-statistics algorithm, the worst-case time complexity of quicksort becomes $\Theta(n \cdot \log(n))$ because the call tree will always be balanced.
- This solution is substantially slower than mergesort, in practice.

2.6 The Convex-Hull Problem

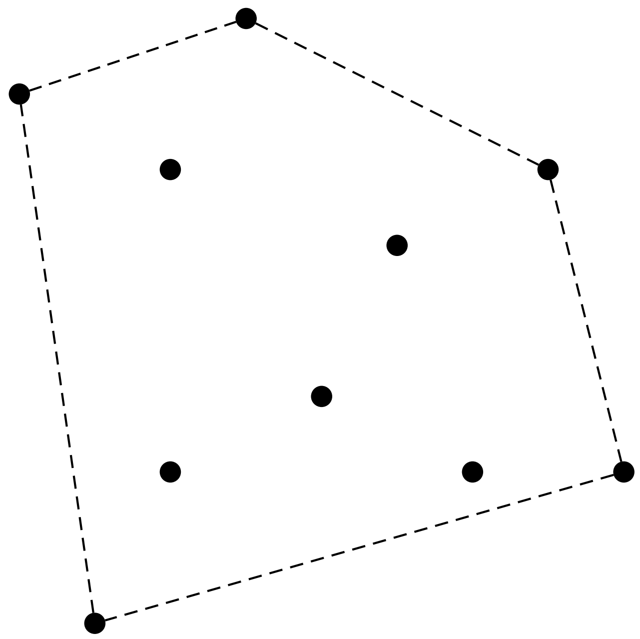
2.6.1 Description of the problem

Given: A finite set of points in a two-dimensional plane.

Find: The smallest convex polygon containing all of the points.

Visualization:

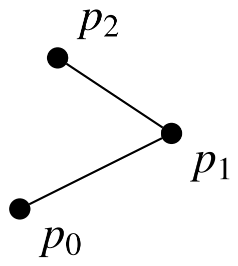
- The "plane" is a wooden board.
- Each point is a nail.
- The containing convex polygon is found by wrapping a rubber band around the nails.



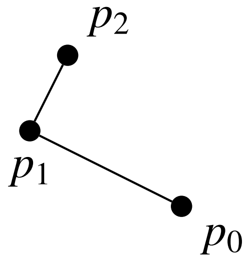
2.6.2 The direction of a turn In the development of algorithms for the construction of convex hulls, it is useful to be able to answer the following:

Question: Given a sequence of three points $\langle p_0, p_1, p_2 \rangle$, which of the following does the line segment connecting them form?

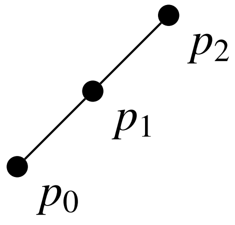
left turn:



right turn:



straight line:



2.6.3 Lemma –Determining the direction of a turn *Let* $p_0 = (x_0, y_0)$*,* $p_1 = (x_1, y_1)$*, and* $p_2 = (x_2, y_2)$ *be points in the plane, and define*

$$\mathsf{Turn}(p_0, p_1, p_2) = (x_1 - x_0) \cdot (y_2 - y_0) - (x_2 - x_0) \cdot (y_1 - y_0)$$

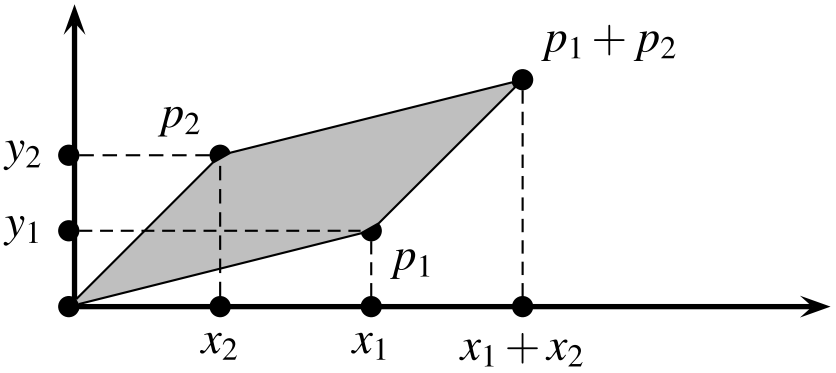
Then $\langle p_0, p_1, p_2 \rangle$ *forms*

(a) *a left turn if* $\mathsf{Turn}(p_0, p_1, p_2) > 0$*;*

(b) *a right turn if* $\mathsf{Turn}(p_0, p_1, p_2) < 0$*;*

(a) *a straight line if* $\mathsf{Turn}(p_0, p_1, p_2) = 0$.

PROOF: First assume that $(x_0, y_0) = (0,0)$, and that x_1, y_1, x_2, y_2 are all nonnegative, and let $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. For $(x_0, y_0) = (0,0)$, $\mathsf{Turn}(p_0, p_1, p_2) = x_1 \cdot y_2 - x_2 \cdot y_1$, and it is easy to see that this value is the shaded area in the diagram below. Indeed,



$$
\begin{aligned}
\mathsf{Turn}(p_0, p_1, p_2) = &\\
& \mathsf{Area}_{\square}((x_2, 0), p_2, p_1 + p_2, (x_1 + x_2, 0))\\
+ \ & \mathsf{Area}_{\triangle}((0,0), p_2, (x_2, 0))\\
- \ & \mathsf{Area}_{\triangle}((0,0), p_1, (x_1, 0))\\
- \ & \mathsf{Area}_{\square}((x_1, 0), p_1, p_1 + p_2, (x_1 + x_2, 0))\\
& = (\tfrac{1}{2} \cdot x_1 \cdot y_1 + x_1 \cdot y_2) + (\tfrac{1}{2} \cdot x_2 \cdot y_2) - (\tfrac{1}{2} \cdot x_1 \cdot y_1) - (x_2 \cdot y_1 + \tfrac{1}{2} \cdot x_2 \cdot y_2)
\end{aligned}
$$

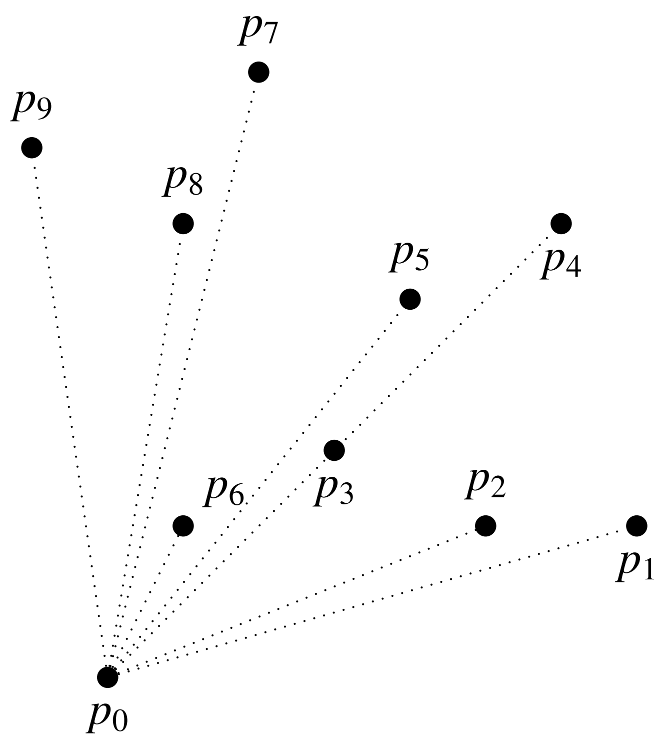
Here $\mathsf{Area}_{\triangle}(-)$ (resp. $\mathsf{Area}_{\square}(-)$) represents the areas of the triangle (resp. trapezoid) with vertices as indicated. As depicted, this area is positive because $\langle p_0, p_1, p_2 \rangle$ defines a left turn. Reversing the rôles of p_1 and p_2 shows that the value is negative for a right turn.

If $p_0 \neq (0,0)$, just translate the whole problem to $\langle q_0, q_1, q_2 \rangle$, with $q_0 = (0,0)$, $q_1 = (x_1 - x_0, y_1 - y_0)$, and $q_2 = (x_2 - x_0, y_2 - y_0)$, and use the above result.

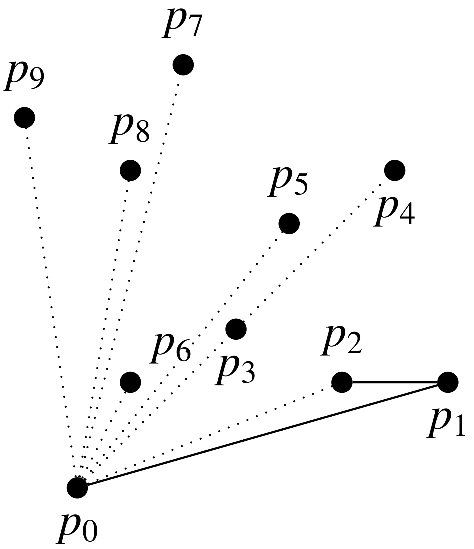
To be complete, it is also necessary to show that this approach still works if some of the coordinates are negative, This is straightforward and omitted. (In the convex-hull problems considered here, all coordinates are nonnegative.) □

2.6.4 The idea of the Graham-scan algorithm

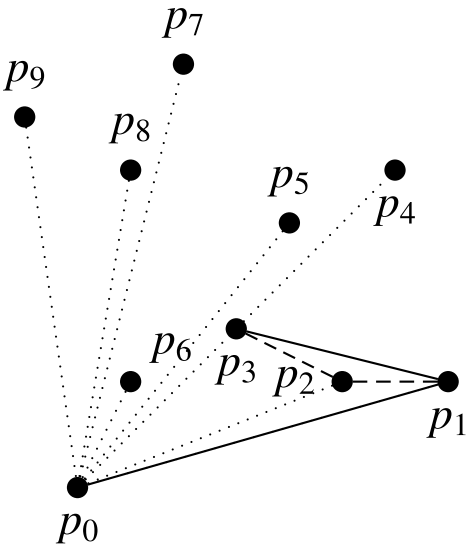
- *Graham scan* is one of the most fundamental algorithms for the construction of a convex hull.
- Even though it is not based upon divide-and-conquer, it may be used as a component in such strategies.
- To begin, call the point with least y value p_0.
- If there is a tie, from the points with least y value chose the one with least x value as well.
- Order the rest of the points according to the angle from p_0, as shown below.
- For points of equal angle, only the one furthest from p_0 need be retained. The others may be discarded.
- In the example below, p_3 *may* be discarded.



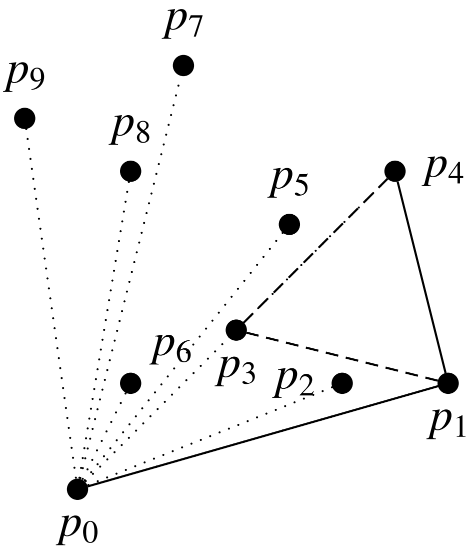
- Begin by connecting the first three points:



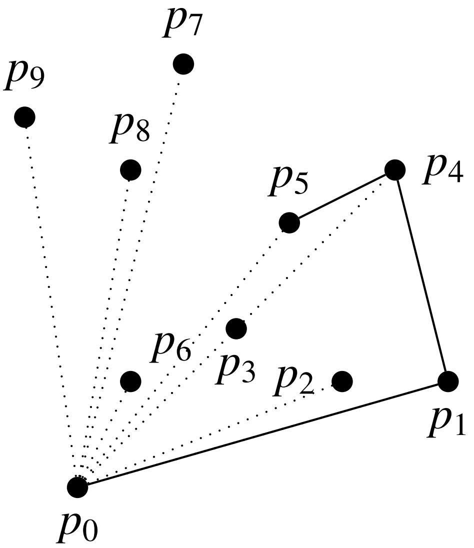
- Next, connect the fourth point p_3.
- Since $\langle p_1, p_2, p_3 \rangle$ forms a right turn, discard p_2.



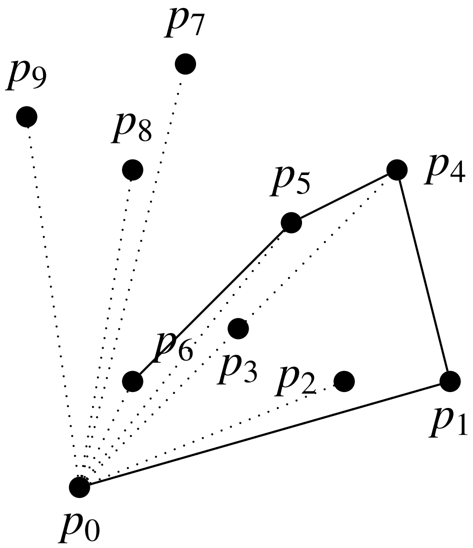
- Now connect p_4.
- Since $\langle p_2, p_3, p_4 \rangle$ forms a right turn, discard p_3.
- Note that p_3 could also have been discarded initially, since it lies on the line from p_0 to p_4.



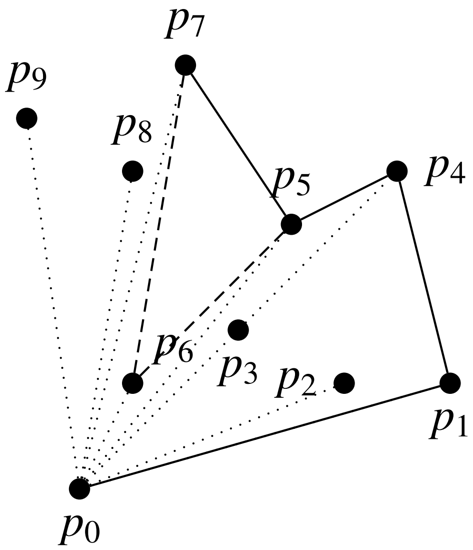
- Now connect p_5.
- Since $\langle p_3, p_4, p_5 \rangle$ forms a left turn, Keep p_4.



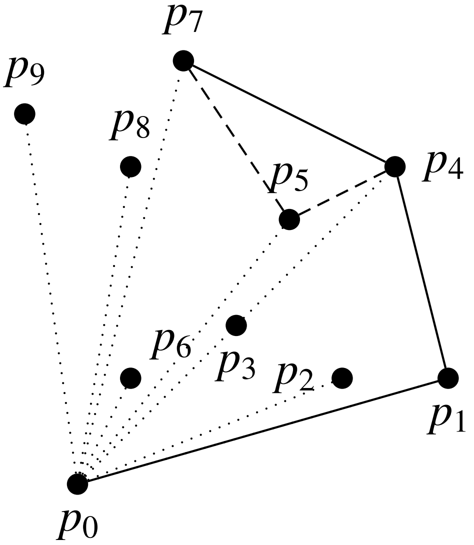
- Now connect p_6.
- Since $\langle p_4, p_5, p_6 \rangle$ forms a left turn, keep p_5.



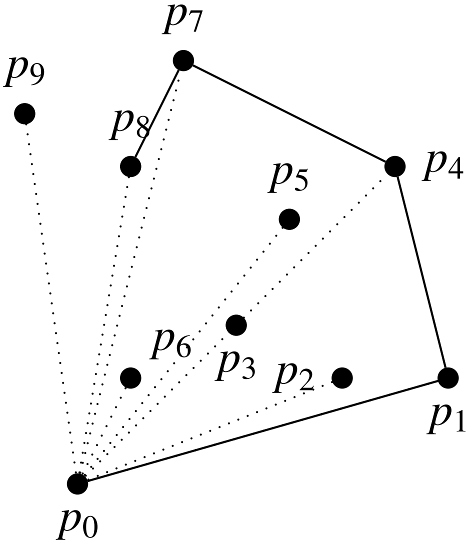
- Now connect p_7.
- Since $\langle p_5, p_6, p_7 \rangle$ forms a right turn, discard p_6.



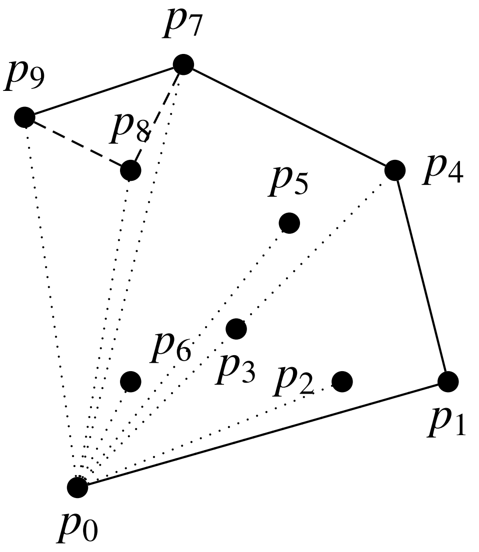
- There is a detail which has been glossed over until now.
- When a right turn is detected, it is actually necessary to check the previous three elements (which are left after the deletion) for a right turn a well.
- Since $\langle p_4, p_5, p_7 \rangle$ forms a right turn, discard p_5.



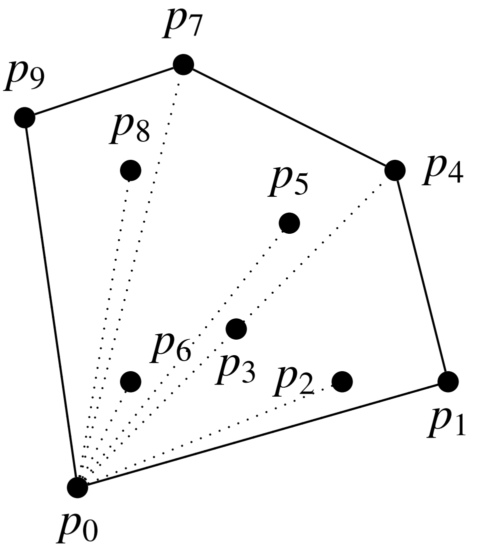
- Since $\langle p_1, p_4, p_7 \rangle$ forms a left turn, the backward search for right turns ends.
- Now connect p_8.
- Since $\langle p_4, p_7, p_8 \rangle$ forms a left turn, keep p_7.



- Now connect p_9.
- Since $\langle p_7, p_8, p_9 \rangle$ forms a right turn, discard p_8.
- Note that this is necessary to check $\langle p_4, p_7, p_9 \rangle$ for a right turn as well.



- The construction is now complete.
- For graphical completeness, p_9 may be connected to p_0, but this is not part of the algorithm.



2.6.5 The high-level algorithm for Graham scan

```
procedure GrahamScan(P : array[0..n] of point ;
                                        ref S : stack of point )
   ⟨ /* The convex hull of P will be returned in S. */
      i, last : 3..n;
      init(S);
      /* The next function does the following: */
      /*   It places the element with least y-coordinate in P[0]. */
      /*   A tie is resolved with least x-coordinate. */
      /*   The rest of the array is sorted wrt angle with P[0]. */
      /*   With equal angles, */
      /*   only the point furthest from P[0] is retained. */
      /*   last identifies the last index of P which is used. */
      sort_by_angle_and_remove_collinear(P, last);
      push(P[0], S);
      push(P[1], S);
      push(P[2], S);
      for i = 3 to last do
          ⟨ while forms_right_angle(next_to_top(S), top(S), P[i])
                  do pop(S);
            push(P[i], S);
          ⟩
   ⟩
```

- The sorting of the points in P may be done using any sorting algorithm.
- $\mathsf{Angle}(p_i) < \mathsf{Angle}(p_j)$ iff $\langle p_0, p_i, p_j \rangle$ forms a left turn.
- p_i and p_j are collinear iff $\langle p_0, p_i, p_j \rangle$ forms a straight line.

2.6.6 Proposition – the complexity of Graham scan *Let S be a set of n points in a two-dimensional plane. The average and worst-case complexity of the Graham-scan algorithm of 2.6.5 for finding the convex hull of S is* $\Theta(n \cdot \log(n))$.

PROOF: The main loop of the algorithm is executed only $\Theta(n)$ times. The dominant item of computation is thus the sorting of the points. In view of the discussion at the end of 2.6.5, the complexity is thus that of sorting. Using mergesort or quicksort, this may accomplished in $\Theta(n \cdot \log(n))$ is worst- and average-case time. □

2.6.7 Divide-and-conquer strategies for the convex-hull problem

- An overall formulation might be as follows:

```
procedure DC_hull(S : set of point ,
                     ref H : convex_hull);
  ⟨ if small(S)
      then H ← solve_directly(S);
       else ⟨ divide(S, S_1, S_2);
              DC_hull(S_1, hull_1);
              DC_hull(S_2, hull_2);
              H ← merge_hulls(hull_1, hull_2);
            ⟩
  ⟩
```

- Within this general formulation, there are two main alternatives:

  The mergesort strategy:

  - The procedure *divide* is trivial.
  - All of the work is done in *merge_hulls*.

  The quicksort strategy:

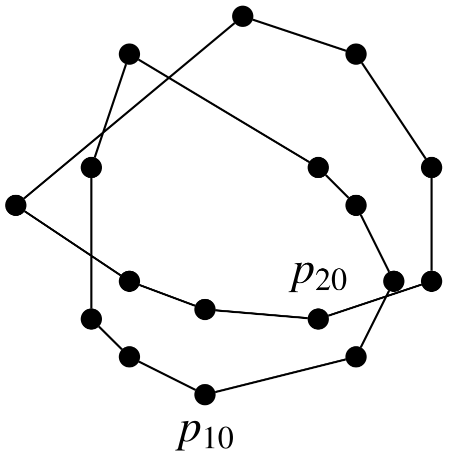
  - The *merge_hulls* procedure is trivial.
  - All of the work is done in *divide*.

- Quicksort-like strategies appear to be more complex than mergesort-like ones.

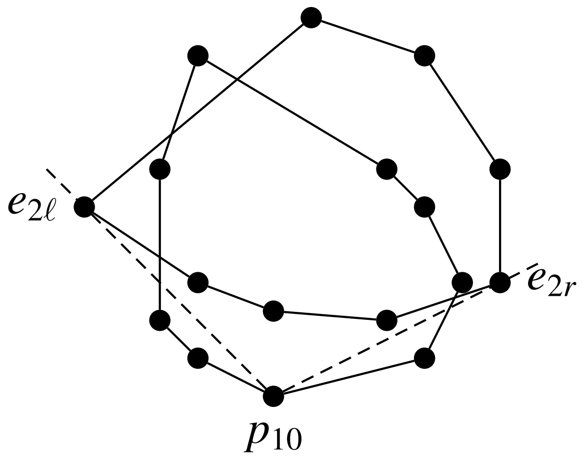
- Here a simple mergesort-like strategy is presented.

2.6.8 A simple strategy for merging two convex hulls

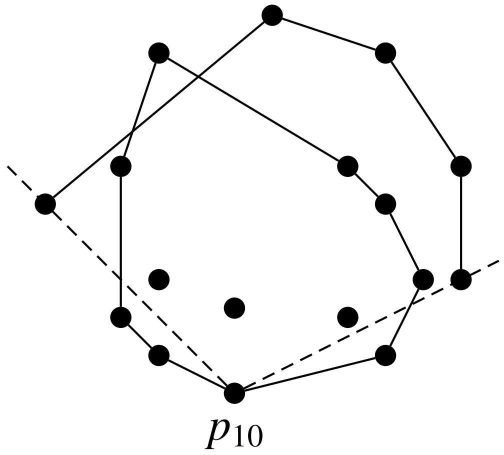
- Given are two convex hulls, with least-y points p_{10} and p_{20}, respectively.
- The goal is to merge them into a single convex hull.
- Assume that the order information (relative to the respective base points p_{10} and p_{20}) is available.



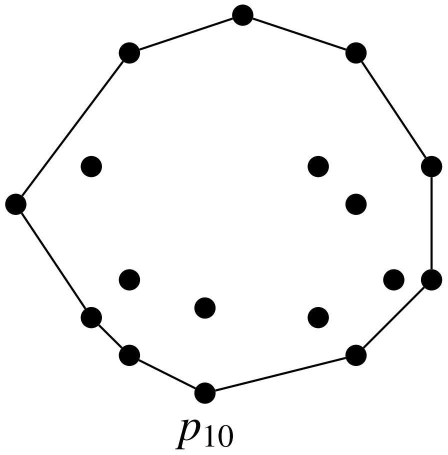
- From the base point p_{10} of the hull with the lesser y value (hull 1) for its least-y point, find the extreme points e_{2r} and $e_{2\ell}$ of the other hull (hull 2) which define the least and greatest angles.



- The nodes “below” these extreme points in hull 2 cannot be part of the combined hull, and may be discarded.
- Note that the remaining points in hull 2 have the same order relative to p_{10} as they did relative to their original base p_{20}.



- The remaining points are combined into a new hull with a modified Graham scan.
- In this modified scan, it is not necessary to sort the nodes from scratch.
- It suffices to merge the two lists containing these points, since each component is already sorted relative to p_{10}.



2.6.9 Complexity analysis of hull merging

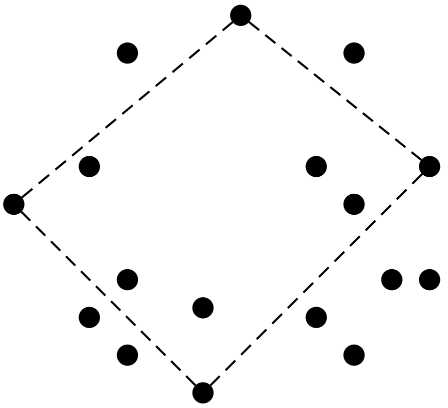
- A full pseudocode description of convex-hull merging is tedious because of the many picky details involved.
- Nonetheless, it is straightforward to characterize the asymptotic complexity.
- Assume that the sizes of $hull_1$ (resp. $hull_2$) is n_1 (resp. n_2), and that $hull_1$ is the one with the smallest "least-y" element.
  - Selecting the hull which has the "least-y" element is clearly $\Theta(1)$, since the hulls are already sorted with their base points in the first position of the respective arrays.
  - Finding the extreme points $e_{2\ell}$ and e_{2r} may be performed in time $\Theta(n_2)$. The naïve approach is simply to compare each point of $hull_2$ with p_{01}.
  - The process of merging the lists of points from the two hulls takes $\Theta(n_1 + n_2)$. This is essentially a standard comparison-based merge.
  - The Graham scan without the initial sort (lines 13-20 of 2.6.5), runs in time $\Theta(n_1 + n_2)$.
- The total complexity is thus $\Theta(n_1 + n_2)$.
- This complexity is valid in all cases (best, worst, average).

2.6.10 The total complexity of divide-and-conquer convex hull

- The relevant recurrence relation is the same as that for mergesort 2.1.2.
- The total complexity is thus $\Theta(n \cdot \log(n))$ in all cases.
- In experimental measurements, this divide-and-conquer approach did not outperform simple Graham scan.

2.6.11 The Floyd-Eddy heuristic

- There is a heuristic which may be used to improve the performance (but not the asymptotic complexity) of many convex hull algorithms.
- The *Floyd-Eddy heuristic* proceeds as follows.



- The four extreme points (least-x, greatest-x, least-y, greatest-y) are identified, and connected to form a quadrilateral. Ties may be resolved arbitrarily.
- Points inside of this box are eliminated from further consideration.
- The time complexity is $\Theta(n)$.