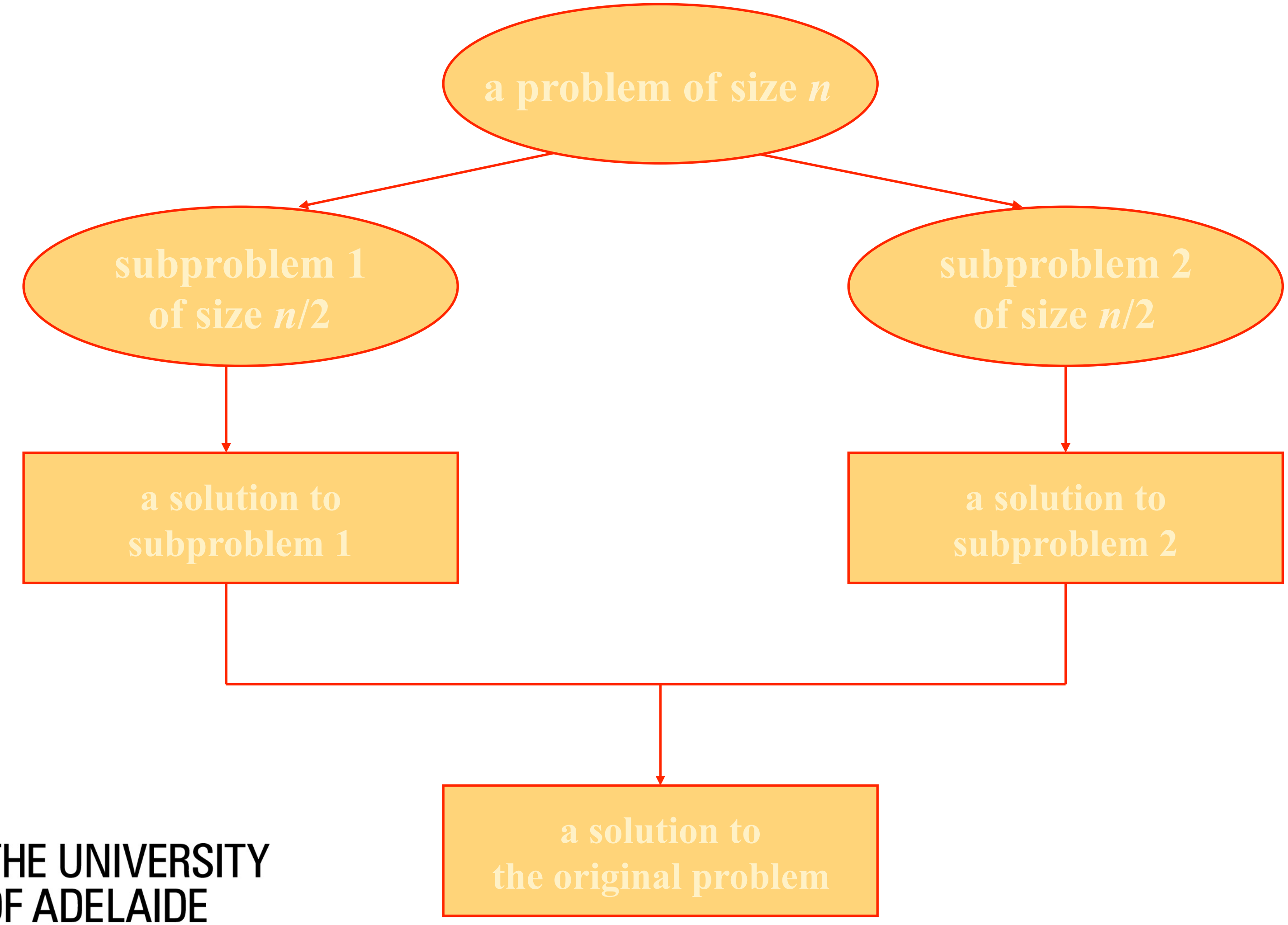
Divide and Conquer

When the sum of the parts is less than the whole

(Levitin Chapter 4)

Divide-and-Conquer Technique (cont.)



Divide-and-Conquer Examples

- Sorting: mergesort and quicksort

- Closest-pair and convex-hull algorithms

General Divide-and-Conquer Recurrence

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

Where:

- $f(n)$ is complexity of the function doing the dividing and combining (always polynomial).
- the problem is divided b ways on each step
- a is the number of sub-problems actually solved on each step.
  - In most divide and conquer algorithms a=b

Mergesort

- Split array A[0..*n*-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
  copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
  copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
  Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
  Mergesort$(C[0..\lceil n/2 \rceil - 1])$
  Merge(B, C, A)

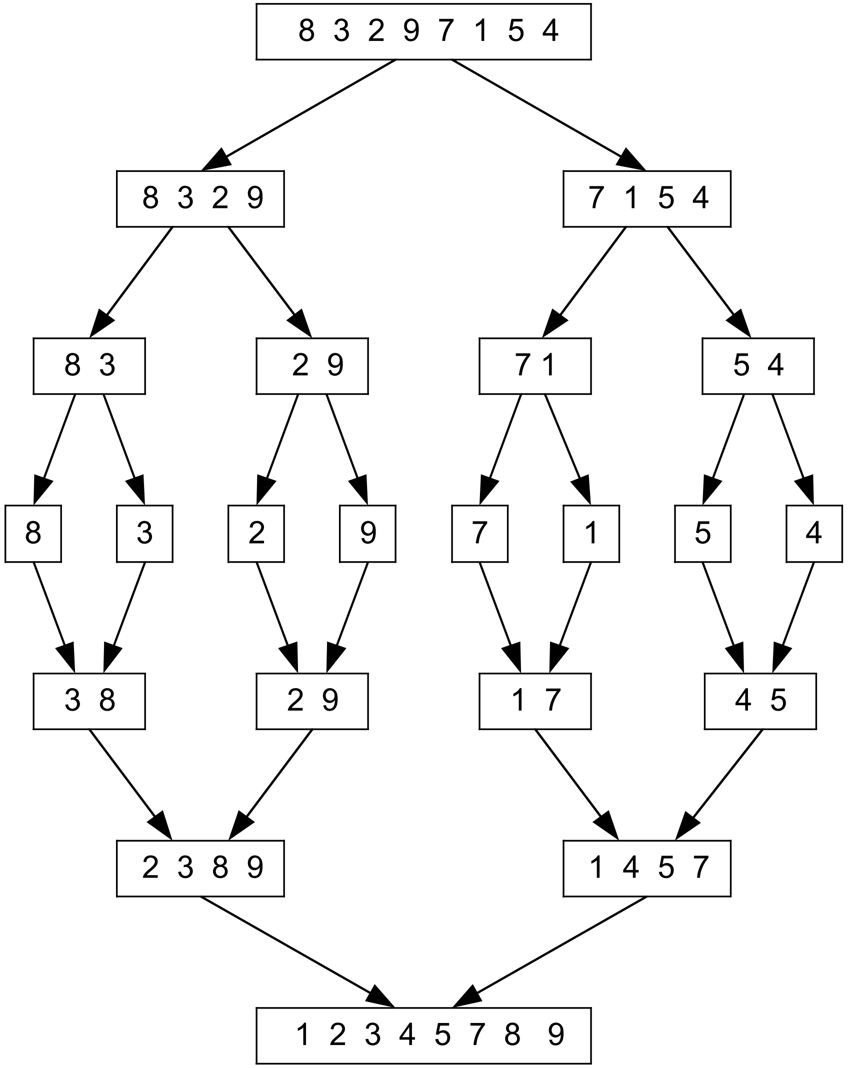
THE UNIVERSITY OF ADELAIDE AUSTRALIA

Pseudocode of Merge

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \leq C[j]$
    $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
  else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
  $k \leftarrow k + 1$
if $i = p$
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Mergesort Example



Analysis of Mergesort

- All cases have same efficiency: $\Theta(n \log n)$

- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

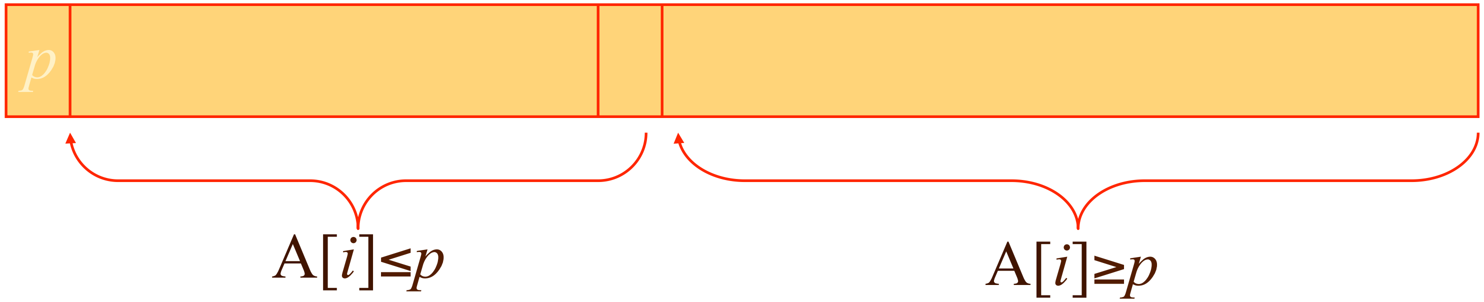
$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

- Space requirement: $\Theta(n)$ (<u>not</u> in-place)

- Can be implemented without recursion (bottom-up)

Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

THE UNIVERSITY OF ADELAIDE AUSTRALIA

Partitioning Algorithm

Algorithm $Partition(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
//   indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
//   this function's value
$p \leftarrow A[l]$
$i \leftarrow l; \quad j \leftarrow r + 1$
repeat
    repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$
    repeat $j \leftarrow j - 1$ **until** $A[j] \cdot p$
    swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Quicksort Example

5 3 1 9 8 2 4 7

Analysis of Quicksort

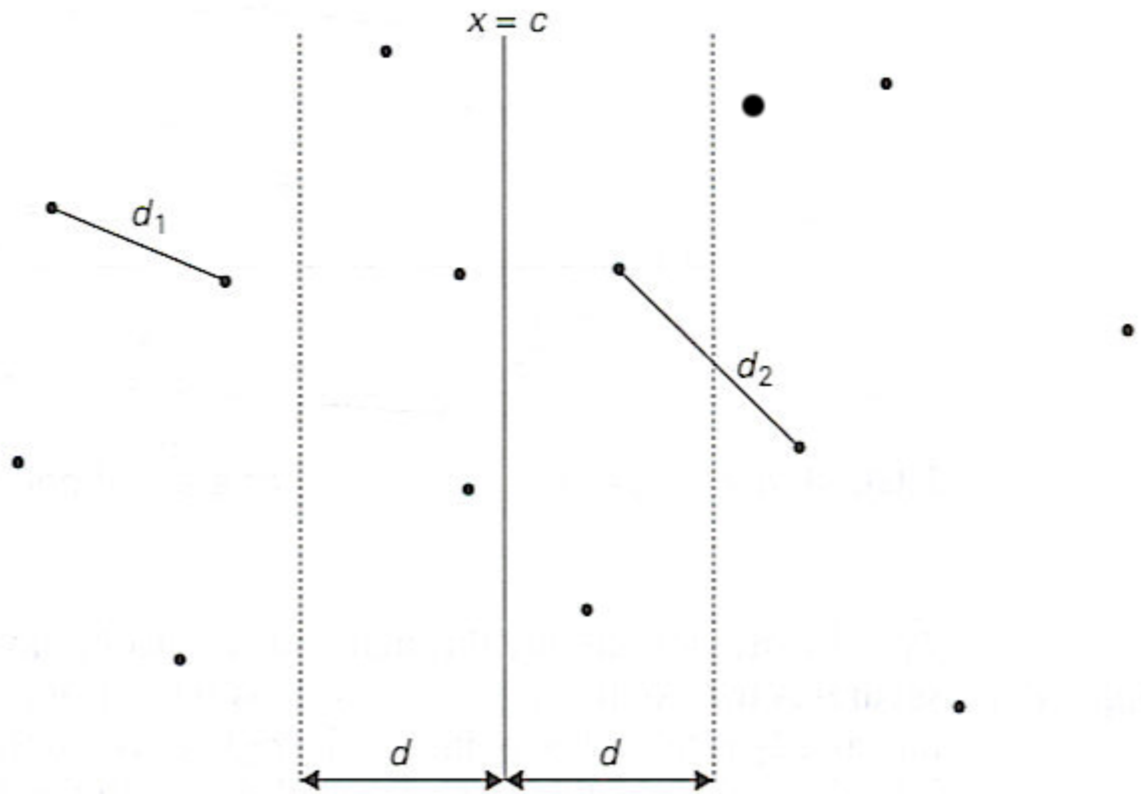
- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$
- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement
- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

THE UNIVERSITY OF ADELAIDE AUSTRALIA

Closest-Pair Problem by Divide-and-Conquer

Step 1 Divide the points given into two subsets S_1 and S_2 by a vertical line $x = c$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.



Closest Pair by Divide-and-Conquer (cont.)

Step 2 Find recursively the closest pairs for the left and right subsets.

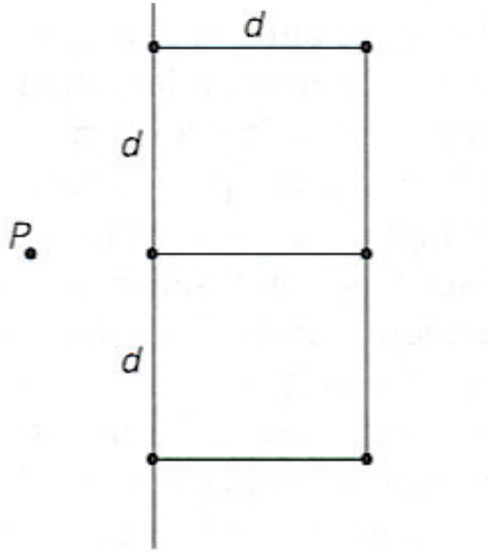
Step 3 Set $d = \min\{d_1, d_2\}$

We can limit our attention to the points in the symmetric vertical strip of width $2d$ as possible closest pair. Let C_1 and C_2 be the subsets of points in the left subset S_1 and of the right subset S_2, respectively, that lie in this vertical strip. The points in C_1 and C_2 are stored in increasing order of their y coordinates, which is maintained by merging during the execution of the next step.

Step 4 For every point $P(x,y)$ in C_1, we inspect points in C_2 that may be closer to P than d. There can be no more than 6 such points (because $d \leq d_2$)!

Closest Pair by Divide-and-Conquer: Worst Case

The worst case scenario is depicted below:



Quick Hull

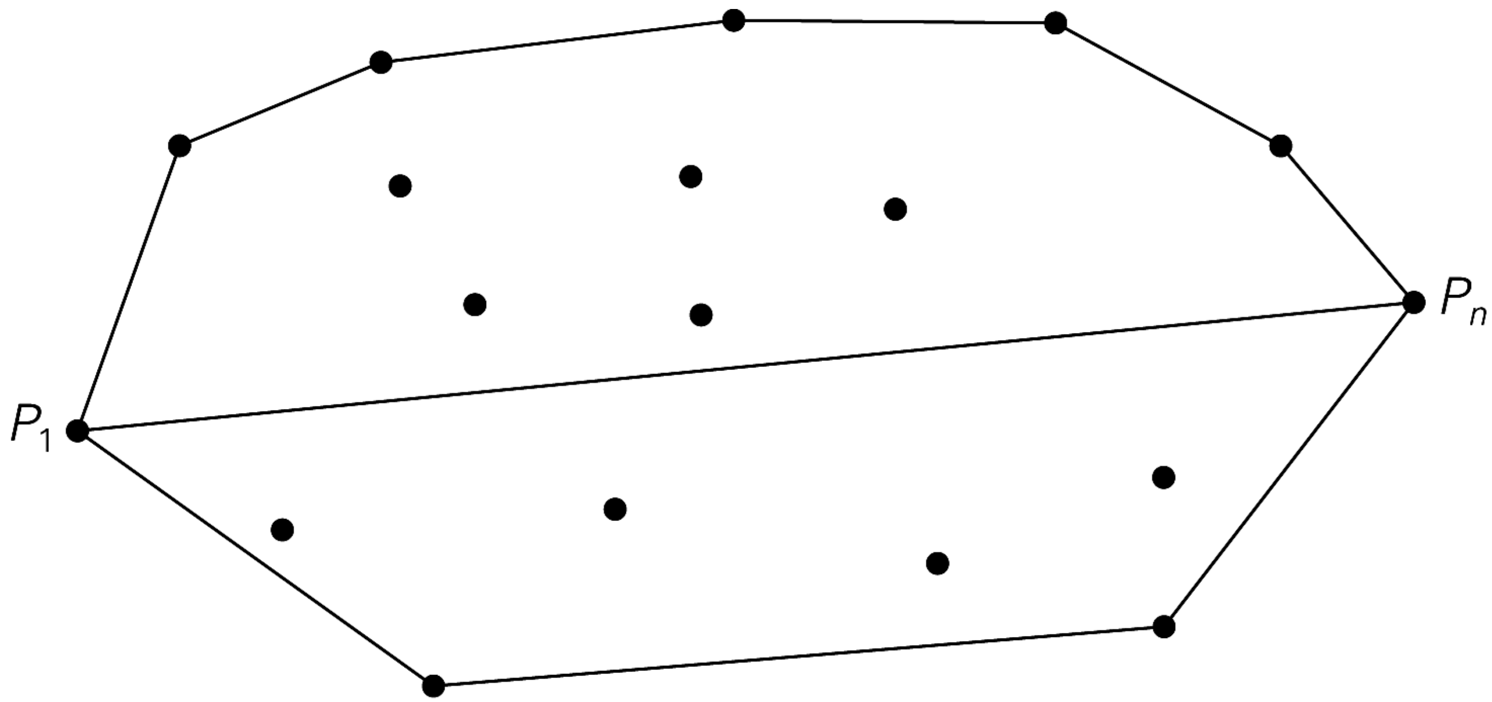


FIGURE 4.8 Upper and lower hulls of a set of points

QuickHull - next -step

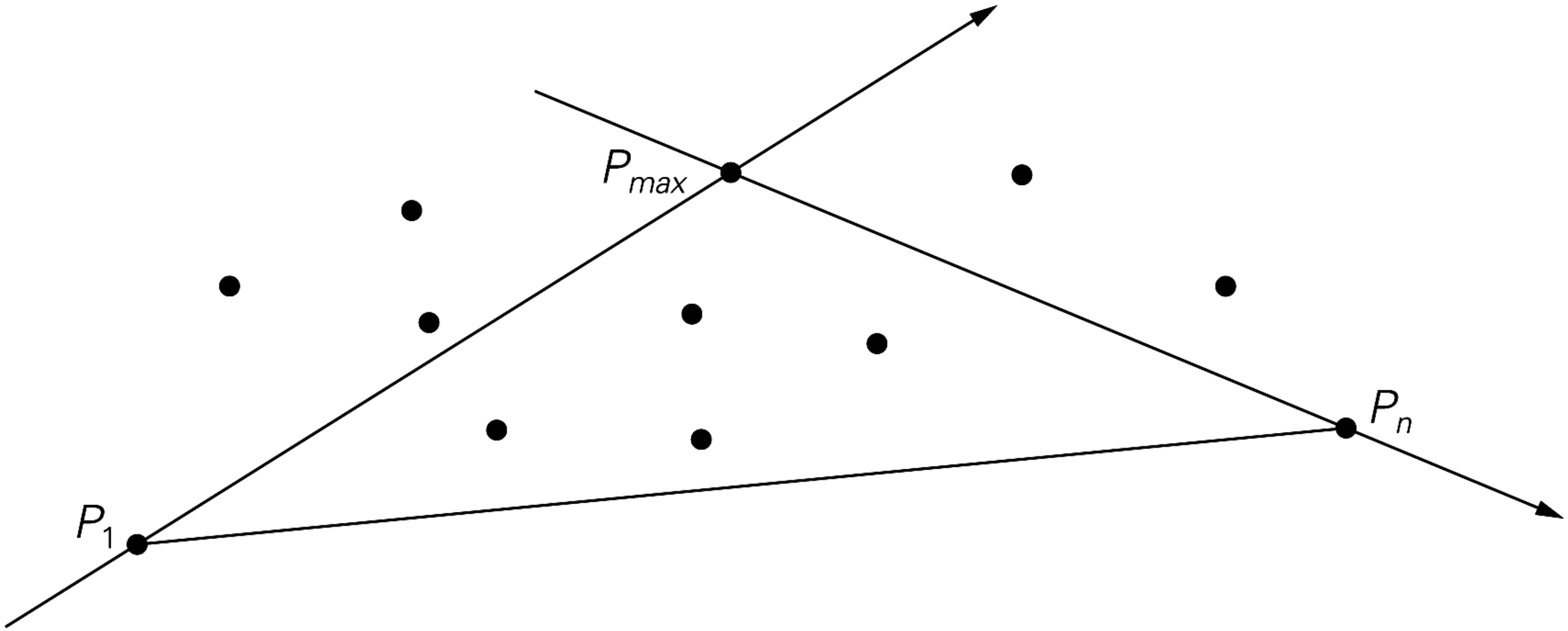


FIGURE 4.9 The idea of quickhull

Decrease and Conquer

Solve a smaller problem on every step

Levitin Chapter 5

Decrease and Conquer

- Often, it is possible to solve a problem by solving a smaller problem first and then using that solution to solve the bigger problem
  - This is the essence of recursive solutions.
- The strategy of decrease and conquer comes in three flavours.
  - Decrease by a constant amount
  - Decrease by a constant factor (usually a factor of two)
  - Decrease by a variable amount.

Decrease by a constant amount

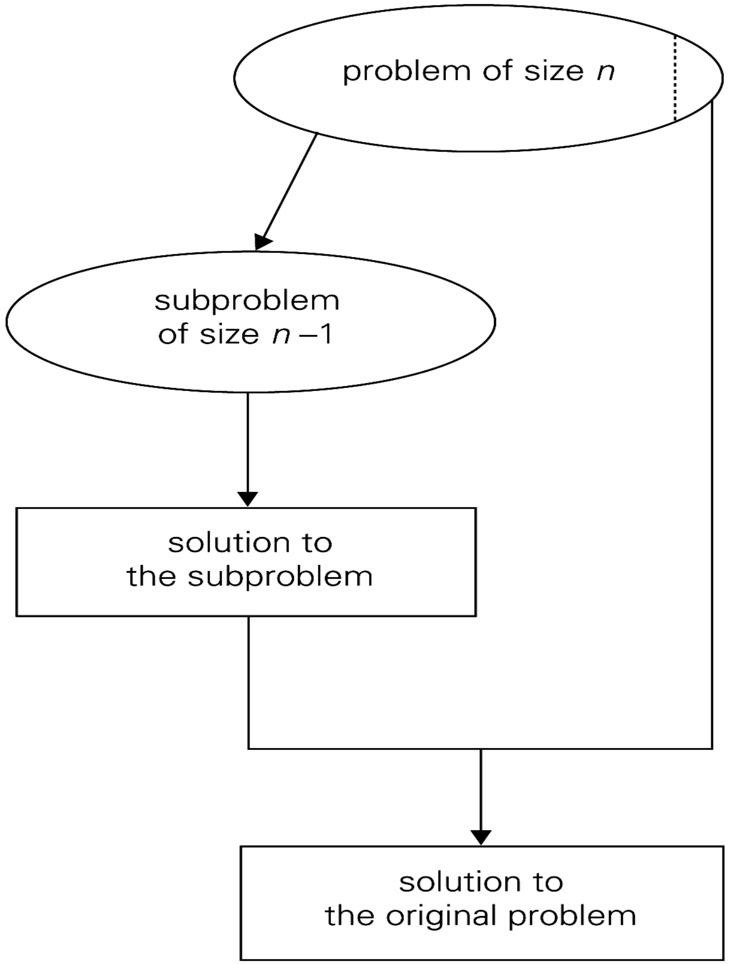


FIGURE 5.1 Decrease (by one)-and-conquer technique

Decrease by a constant factor

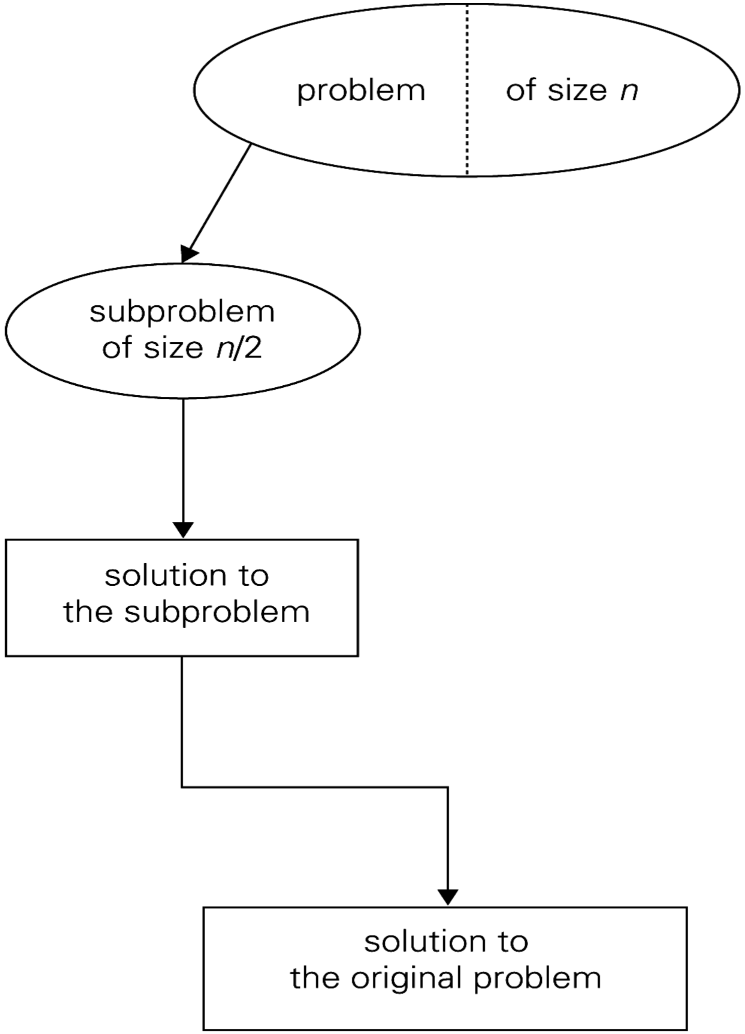


FIGURE 5.2 Decrease (by half)-and-conquer technique

Decrease by a constant amount

- Examples
  - Generating permutations
  - Generating subsets

Generating Permutations

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 21 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

FIGURE 5.12 Generating permutations bottom up

- Adv - simple
- Disadv - requires insertion and also takes a long time to get first perm.

Better permutations - Johnson-Trotter

- Start with the list you want to generate
  - e.g. 1 2 3 4 5 6 7 8
- Assign every element a direction arrow
  - <1<2<3<4<5<6<7<8
- A number is mobile if it points to a smaller number ajacent to it
  - In the above list all numbers except 1 is mobile
  - <8 is the largest mobile number.

THE UNIVERSITY OF ADELAIDE AUSTRALIA

Johnson-Trotter

- Algorithm - start with a sorted permutation with all arrows pointing backwards.

While the latest permutation has a mobile element

  find the largest mobile element k

  swap k and the integer it points to

  now reverse the direction of all the elements larger than k

  add this permutation to the list

End while

- Alg finishes with the permutation
- <1<23>4>…n>

Generating subsets

n	**subsets**							
0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

FIGURE 5.13 Generating subsets bottom up

A better subset generating algorithm

- The last algorithm required a lot of storage.
- We can generate a stream of subsets easily by using a mapping between binary numbers and set membership.
- e.g.

  000 -> {}, 001-> $\{a_3\}$, 010 -> $\{a_2\}$, 011-> $\{a_2,a_3\}$, 100 -> $\{a_1\}$, 101 -> $\{a_1,a_3\}$ ….

THE UNIVERSITY OF ADELAIDE AUSTRALIA

Decrease by a constant factor

- Examples
  - Find the fake coin
  - Russian peasant Method for Multiplication
  - Binary search

Find the fake coin

- You have a pile of n coins - one is a fake
- The fake is lighter
- You have an old-fashioned set of scales
- Find the fake coin quickly.

Algorithm

- If n is odd.
  - Put one coin aside, divide the remaining coins into two piles and compare them on the scales.
  - If they are both equal the fake is the coin you put aside.
  - Otherwise take the lighter set of coins and repeat the algorithm.
- If n is even
  - As above but don't put a coin aside.

Russian peasant multiplication

- To mulitply n my m
  - If n is even
    - Halve n and double m
  - If n is odd
    - Add m to our total
    - Halve (n-1) and double m
- Repeat
- This method is very fast conventional computers
  - Why?

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1, 040	
1	2, 080	(+1040)
	2, 080	$+(130 + 1040) = 3, 250$

(a)

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1, 040	1, 040
1	2, 080	2, 080
		3, 250

(b)

FIGURE 5.14 Computing $50 \cdot 65$ by multiplication à la russe

Reduce by a variable amount

- Examples
  - Finding the median
  - Euclids algorithm.

Finding the median

- The median is the middle-ranked value in a set of numbers.
- Trivial algorithm…
  - Sort the list of numbers
  - Go halfway along the sorted list
    - That is our median value
  - O(n log n) Not the most efficient solution if we only want the median.

Fast median

- Use quicksort-style partitioning.
  - Pick a partition pivot element: p
  - Partition the list into a list L whose elements are less than p and a list H whose elements are greater than p.
  - If the length of L > length of H then
    - Recursively search for the median in L
  - Else, if the length of H > length of L
    - Recursively search for the median in H
  - Else length of H == length of L, p is the median. Stop.
- Question… what is the average time-complexity of this algorithm?

Euclids algorithm

```
// finds gcd(m,n)
While n ≠ 0
    r = n mod m
    m = n
    n = r
Return m
```

- Notice how both n and m shrinking by a variable amount
  - Relies on identity: gcd(m,n) = gcd(n,m mod n)

THE UNIVERSITY OF ADELAIDE AUSTRALIA

When to use

- Use divide and conquer when the cost of doing the sub-problems + the cost of dividing and combining is less than the cost of doing the whole thing.
  - Divide and conquer also has big advantages in parallel applications.
- Use decrease and conquer when:
  - The solution arises naturally
  - Overheads are low