ALGORITHM DESIGN

Algorithm types:

- Brute Force ✓
- Greedy algorithms ✓
  - * Scheduling ~
  - * Bin packing ✓
- Divide & Conquer ~
  - * Merge Sort ~
  - * Closest-Points problem ~
- Dynamic programming
  - * Coin change ✗
  - * Fibonacci ✓
- Randomized algorithms ~
- BackTracking
  - * Maze solving ~
  - * Permutations ✗

• BRUTE FORCE

Straightforward of solving a problem that rely on sheer computer power and trying every possibility rather than advanced techniques to improve efficiency

↳ Almost always quadratic (often exponential)

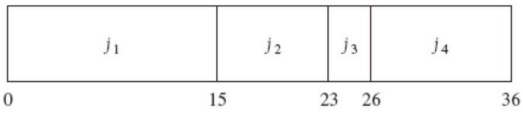
• GREEDY ALGORITHM

- Always choose the locally best solution → called greedy cause you do what is best at the moment
- Often a bad idea, but does work sometimes
- Some of the most used graph algorithms are greedy algorithms

* Scheduling

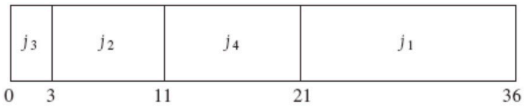
Job	Time
j_1	15
j_2	8
j_3	3
j_4	10

First Come First Serve (FCFS)



Average time 25

Figure 10.2 Schedule #1



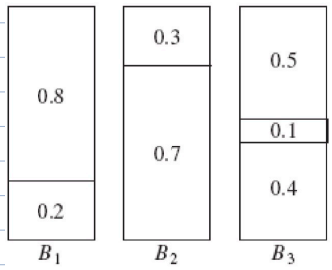
Average time 17,75

Figure 10.3 Schedule #2 (optimal)

* BIN PACKING

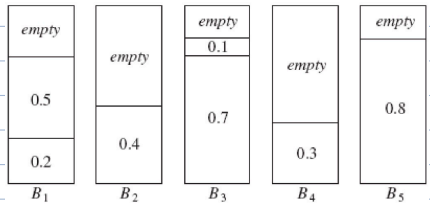
↳ Given n items with B_1, B_2, etc. Such that $0 \leq B_i \leq 1$ for $0 \leq i \leq n$, pack them into the fewest number of unit capacity bins

- On/Off line

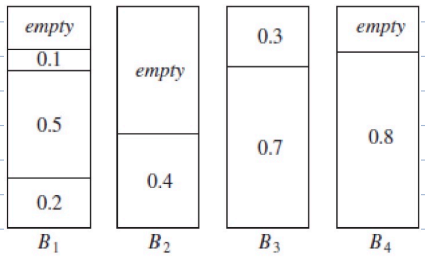


- Next fit

If current item fits in current bin, place it there, otherwise start new bin



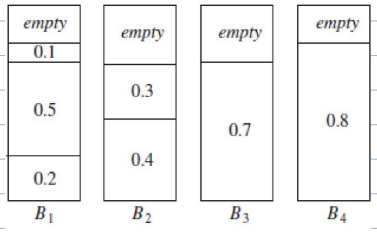
- Best fit



New items placed in a bin where it fits the tightest. If it does not fit in any bin, then start a new bin.

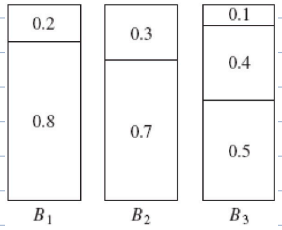
↳ Can be implemented in $O(n \log n)$ time, by using a balance binary tree storing bins ordered by remaining capacity

- First fit



Scans the bins in order and place the new item in the first bin that is large enough to hold it. A new bin is created only when an item does not fit in the previous bins.

- First fit decreasing (off-line)



We know the sizes of all items in advance, a natural solution is to first sort items by size, from largest to smallest, and then apply:
- First Fit Decreasing
- Best Fit Decreasing

• DIVIDE & CONQUER

- Related to recursion
- The idea is to split the problem in smaller problems, that (recursively) might be solved with the same algorithm as the main problem.
- Examples are MergeSort, QuickSort, Towers of Hanoi.
- It can utilize a parallel platform (multiprocessor, multi core systems).

* MergeSort

- Time Complexity: Not a trivial task (due to recursion)
  For each recursive step: $t(n) = 2t(n/2) + O(n)$ → generalized to: $t(n) = a \cdot t(n/b) + f(n)$

- The Master Theorem:

REVIEW THIS!!

* if $f(n) = O(1)$
  $t(n) = at(n/b) + f(n)$

* If $f(n) = O(n)$
  $t(n) = at(n/b) + f(n)$

- The master theorem recognizes 2 cases:
  1. If $a=1$ and $b>1$ → $t(n) = O(\log n)$
  2. If $a>1$ and $b>1$ → $t(n) = O(n^{\log_b a})$

- The master theorem recognizes 3 cases:
  1. If $a > b$ → $t(n) = O(n^{\log_b a})$
  2. If $a = b$ → $t(n) = O(n \log n)$
  3. If $a < b$ → $t(n) = O(n)$

• Application of divide & conquer

REVIEW THIS !!

▪ Closest-Points problems

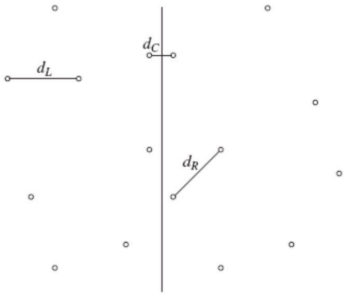


Figure 10.30 P partitioned into P_L and P_R; shortest distances are shown

- We can draw an imaginary vertical line that partitions the point set into two halves, P_L and P_R.
- Either the closest points are both in P_L, or both in P_R, or one in P_L and other in P_R (this distances are d_L, d_R and d_C
- We can compute d_L and d_R recursively. The problem then is to compute d_C
- Since we would like an $O(N \log N)$, we must be able to compute d_C with only $O(N)$

Two half-sized recursive calls and $O(N)$ additional work $\rightarrow O(N \log N)$

- $\delta = \min(d_L, d_R)$ → We only need to compute d_C if d_C improves on δ.

If d_C is such a distance, then the 2 points that define d_C must be within δ of the dividing line (we'll define this area as strip)

This observation limits the number of points that need to be considered. (in our case $\delta = d_R$)

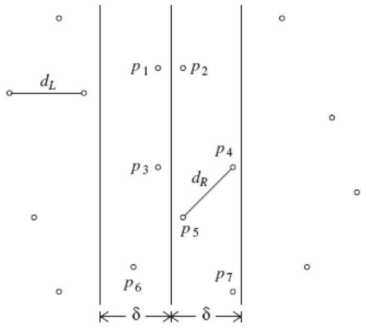


Figure 10.31 Two-lane strip, containing all points considered for d_C strip

- Two strategies to compute d_C:

- For large point sets that are uniformly distributed, the number of points that are expected to be in the strip is very small. Indeed, only $O(\sqrt{N})$ points are in the strip on average

```
// Points are all in the strip
for (i=0; i < numPointsInStrip; i++)
    for (j=i+1; j < numPointsInStrip; j++)
        if (dist(Pi, Pj < δ)
            δ = dist(Pi, Pj)
```

→ Brute-force calculation of $\min(\delta, d_C)$

In the worst case, all the points could be in the strip, so this strategy does not always work in linear time.

We can improve this algorithm with the following observation:

The y coordinates of the two points that define d_C can differ by at most δ. Otherwise, $d_C > \delta$. Suppose that the points in the strip are sorted by their y coordinates. Therefore, if p_i and p_j's y coordinates differ by more than δ, then we can proceed to $p_i + 1$.

```
→ // Points are all in the strip and sorted by y-coordinate
for (i=0; i < numPointsInStrip; i++)
    for (j=i+1; j < numPointsInStrip; j++)
        if (pi and pj's y-coordinate differ by more than δ)
            break; // Go to next pi
        else
            if (dist(Pi, Pj) < δ)
                δ = dist(Pi, Pj);
```

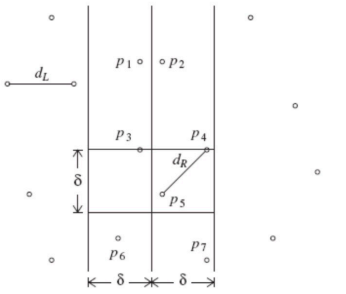


Figure 10.34 Only p_4 and p_5 are considered in the second for loop

→ Shows, for instance, that for point p_3, only the two points p_4 and p_6 lie in the strip within δ vertical distance

↳ This is because these points must lie either in the δ by δ square in the left half of the strip.

On the other hand, all the points in each δ by δ square are separated by at least δ.

In the worst case, each square contains four points, one at each corner. One of these points is p_i, leaving at most seven points to be considered.

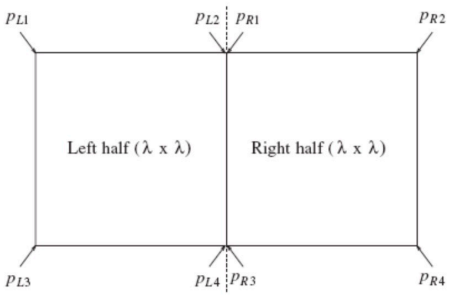


Figure 10.35 At most eight points fit in the rectangle; there are two coordinates shared by two points each

- Notice that even though p_{L2} and p_{R1} have the same coordinates, they could be different points. It is only important that the number of points in the λ by 2λ rectangle be O(1)

- Because at most seven points are considered for each p_i, the time to compute a d_c that is better than δ is O(N). Thus, we appear to have an O(N log N) solution, but not yet.

- The problem is that we have assumed that a list of points sorted by y-coordinate is available. If we perform this sort for each recursive call, then we have O(N log N) extra work: this gives an $O(N \log^2 N)$. However it's not hard to reduce the work for each recursive call to O(N), ensuring an O(N log N) algorithm.

- We will maintain two lists. One is the point list sorted by x-coordinate (P), and the other is the point list sorted by y-coordinate (Q). These can be obtained by a preprocessing sorting step at cost O(N log N) and thus does not affect the time bound. P_L and Q_L are the lists passed to the left-half recursive call, and P_R and Q_R are the lists passed to the right-half recursive call. We have already seen that P is easily split in the middle. Once the dividing line is known, we step through Q sequentially, placing each element in Q_L or Q_R as appropriate. It is easy to see that Q_L and Q_R will be automatically sorted by y-coordinate. When the recursive calls return, we scan through the Q list and discard all the points whose x-coordinates are not within the strip. Then Q contains only points in the strip, and these points are guaranteed to be sorted by their y-coordinates.

• RANDOMIZED ALGORITHMS (AKA MONTE CARLO ALGORITHMS) REVIEW THIS!!

- A choice in a Monte Carlo algorithm is made randomly
- They are surprisingly good for solving some optimization problems
- Quick Sort has elements of randomness which ones?
- Can be used in primality testing
- Randomness be used to counter "anti solver algorithm" eg. in sudoku solvers.

• DYNAMIC PROGRAMMING ALGORITHMS

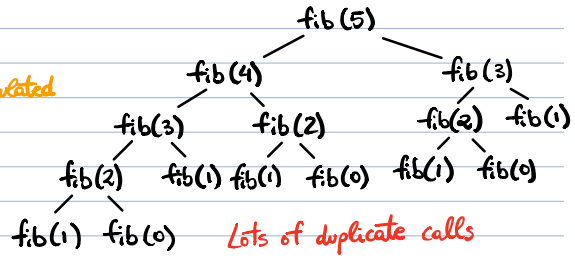
- "Never do the same calculation twice"
- In dynamic programming, the result is stored in a table every time a calculation is done / a subproblem is solved.
- If the same calculation / problem solving is later needed it is just a matter of table lookup.
- It can be used to repair the recursive version of the Fibonacci problem.

* Fibonacci and dynamic programming

- Dynamic programming is useful when sub-problems share sub-sub-problems:
  - → Fib(8) has the sub-problems Fib(7) and Fib(6)
  - → Fib(7) and Fib(6) both has the sub-sub-problem Fib(4)

⇒ Dynamic programming solves each sub-problem once, and stores the answer somehow for later re-use.

```
/** Recursive method to calculate Fibonacci numbers
    pre: n >= 1
    @param n The position of the Fibonacci number being calculated
    @return The Fibonacci number
public static int fibonacci(int n) {
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```



What if we stored/reused values?

```
public static HashMap<Integer, Long> map;
public static long fib(int a){
    if(map.get(a) != null){
        return map.get(a);
    }
    if(a <= 0) return 0L;
    if(a == 1) return 1L;

    long res = fib(a-1) + fib(a-2);
    map.put(a, res);
    return res;
}
```

a	Fib(a)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34

- Without dynamic programming:
  Fib(8) → 67 calls

- With dynamic programming:
  Fib(8) → 15 calls

• BACKTRACKING ALGORITHMS

- Used to remember earlier states of the problem solving, so it is possible to go back and restart the problem solving from this point
- In every step a choice has to be made before doing the next step
- A solving attemp ends as soon as it is clear, that it will not lead to a solution.

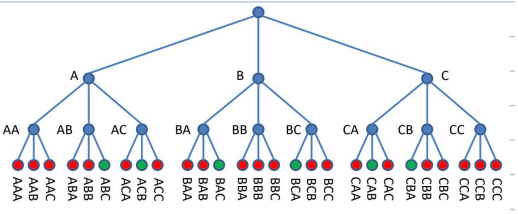
* Maze solving

Problem - generate all permutations of the three letters A, B and C → Solution : ABC BCA ACB CAB BAC CBA

Brute-force algorithm:

```
char c[] = new char[3];
...
for(c[0] = 'a'; c[0] <= 'c'; ++c[0])
    for(c[1] = 'a'; c[1] <= 'c'; ++c[1])
        for(c[2] = 'a'; c[2] <= 'c'; ++c[2])
            if(c[0] != c[1] && c[0] != c[2] && c[1] != c[2])
                System.out.println(" " + c[0] + c[1] + c[2]);
```

Brute-force tree

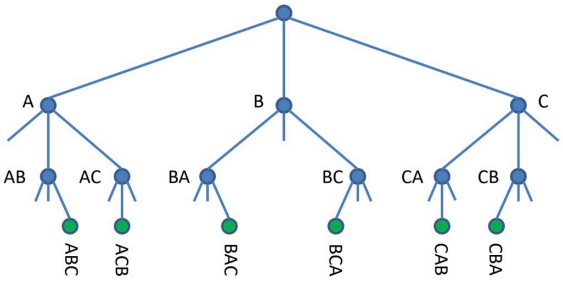


Stopping useless attempts early

```
char c[] = new char[3];
...
for(c[0] = 'a'; c[0] <= 'c'; ++c[0])
  if(Ok(0))
    for(c[1] = 'a'; c[1] <= 'c'; ++c[1])
      if(Ok(1))
        for(c[2] = 'a'; c[2] <= 'c'; ++c[2])
          if(Ok(2))
            if(c[0] != c[1] && c[0] != c[2] && c[1] != c[2])
              System.out.println(" " + c[0] + c[1] + c[2]);
```

```
...
boolean Ok(int position){
    for(int i = 0; i < position; i++)
        if(c[i] == c[position])
            return false;
    return true;
}
```

Reduced tree



Backtracking version:

```
char c[] = new char [3];
TryPosition (0);
...
void TryPosition (int position){
    if ( position == 3)
        System.out.println (" " + c[0] + c[1] + c[2]);
    else
      for ( c[position] = 'a' ; c[position] <= 'c' ; ++ c[position] )
        if (Ok(position))
          TryPosition (position +1);
}
...
```

```
...
boolean Ok (int position){
  for (int i=0 ; i < position ; ++i)
    if (c[i] == c[position])
      return false ;
  return true;
}
```

General backtracking template

```
TryStep(step) {
  if (done(step))
    ReportResult();
  else
    for all possible actions
      if (Ok(action)){
        DoAction();
        TryStep(next(step));
        undoAction();
      }
}
```

• HEURISTIC ALGORITHMS

- Many of the previous algorithm types might be imparved using heuristics, which means that in each step "hints" will be used to make better choices
- The heuristics could come from guessing, analyzing, experience, statistics, etc.
- Heuristics are often used to prioritize the choices in backtracking.
- Heuristics can help solve exponential problems in almost polynomial time.