Introduction to Algorithms

Meng-Tsung Tsai

10/29/2019

Announcements

Written Assignment 2 is due by Oct 31, 15:40. at https://e3.nctu.me

Programming Assignment 2 is due by Nov 5, 23:59. at https://oj.nctu.me

Midterm will be held in class on Nov 05 from 10:10 - 12:30.

Scope: slides 01 - 12, assignments, and their generalizations.

No office hour this Wednesday (I have a talk on that day). I am free on Thursday 10:00-11:00 and Friday 10:30-11:30. If you have questions to ask me, please drop by my office EC336 then.

About Midterm

2 problemsets about asymptotic bounds: was1-p1, was1-p2, was2-p1, quiz1-p2, quiz1-p3.
$\geq$ 2 problemsets about DP (one is monotonic path): was2-p2, was2-p3, was-p4, pas2-p1, quiz1-p4.
$\geq$ 1 problemset about reduction: was1-p5, was2-p6, quiz1-p5.

$\geq$ 1 problemset about greedy algorithms: slides 12.

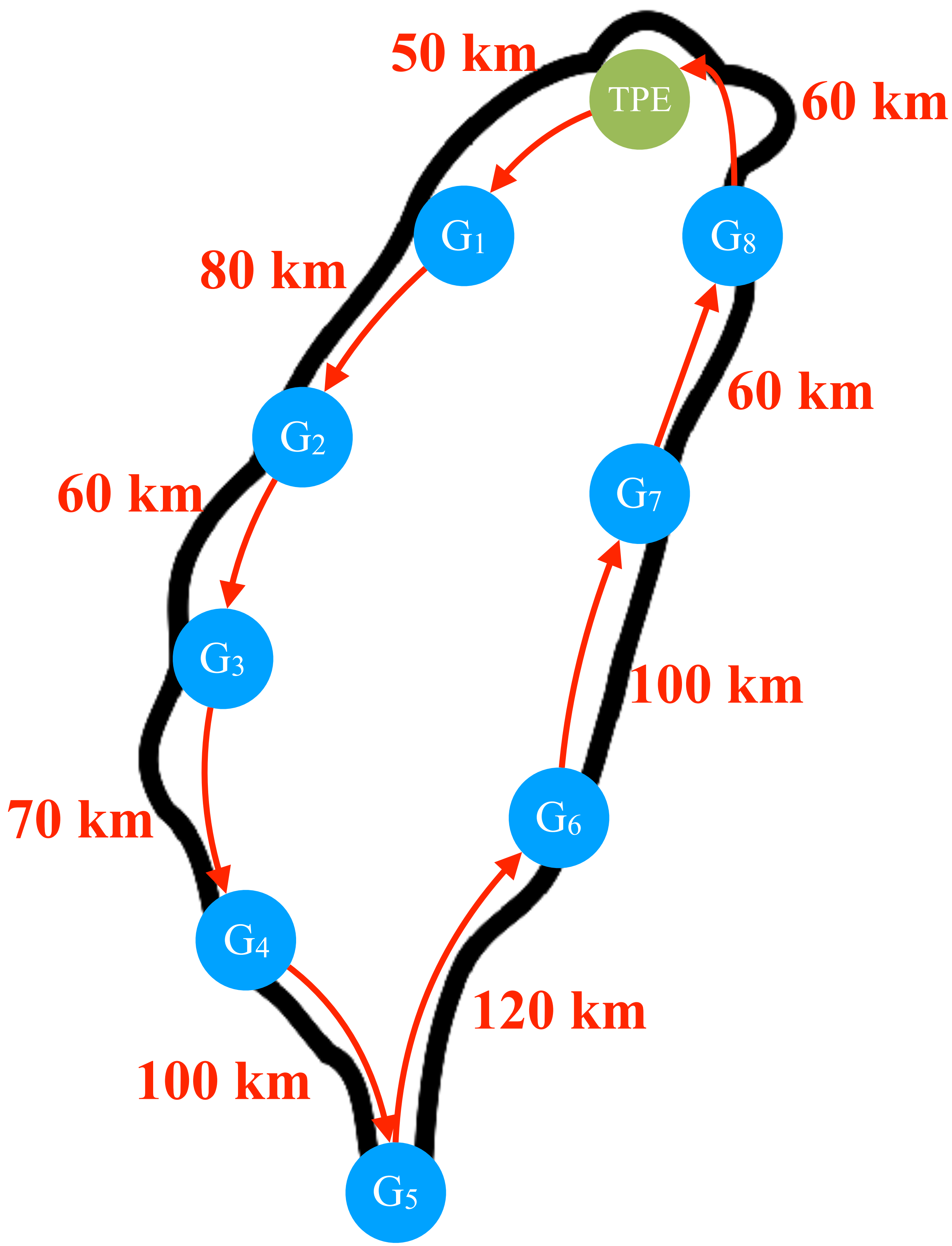
$\geq$ 1 problemset about geometry algorithms: slides 6, 7.

Don't forget the "I don't know" policy.

You may bring four cheating sheets in A4 size.

Greedy Algorithms

Pumping Gas Problem

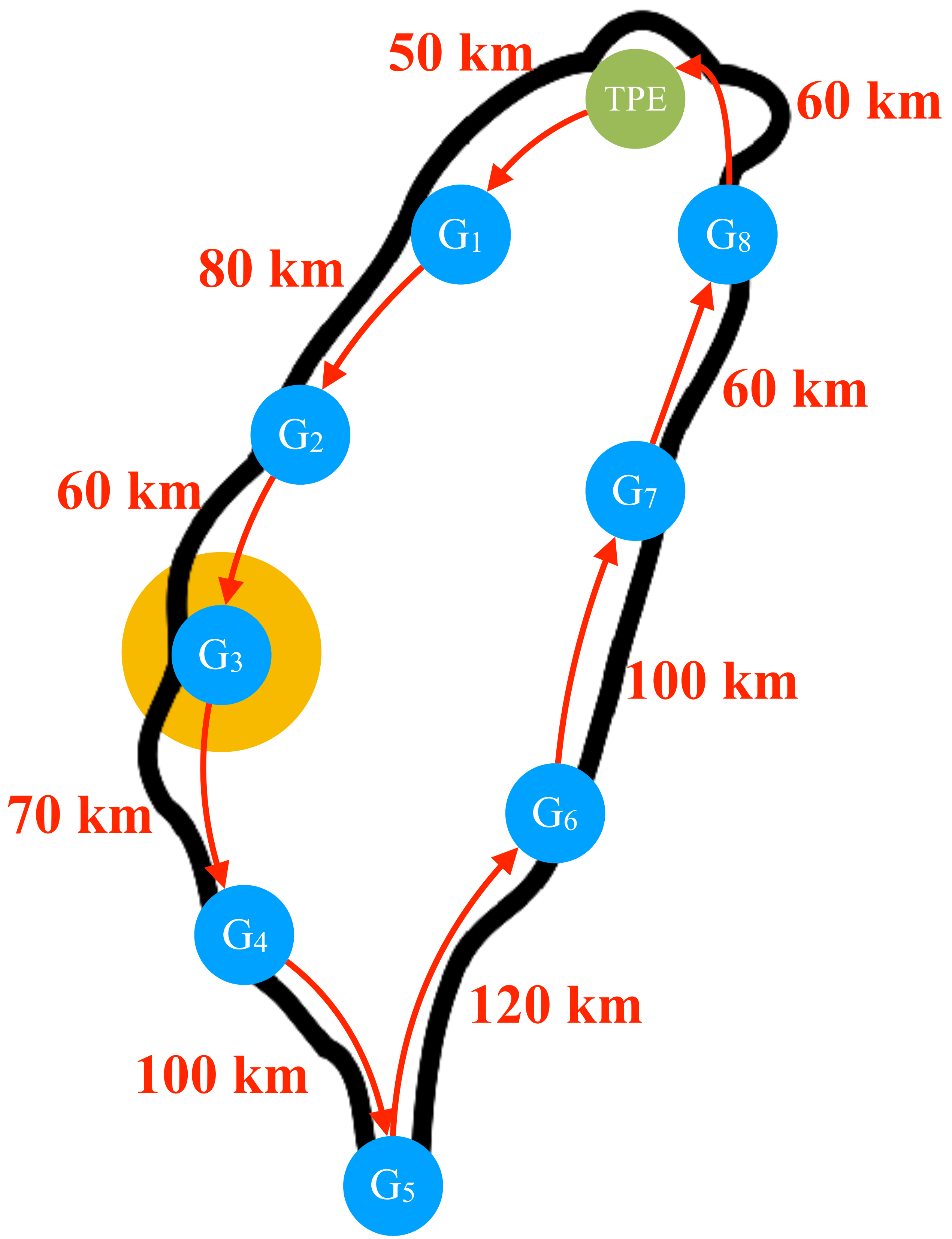


Alice has an old car. Once the car gets filled up, it can run for 200 km without any stop.

Alice would like to drive around Taiwan counterclockwise starting from Taipei. Her car is filled up in Taipei, and can be refilled at any Gi along the route.

Goal: minimize the number of stops to pump gas in the tour.

Pumping Gas Problem

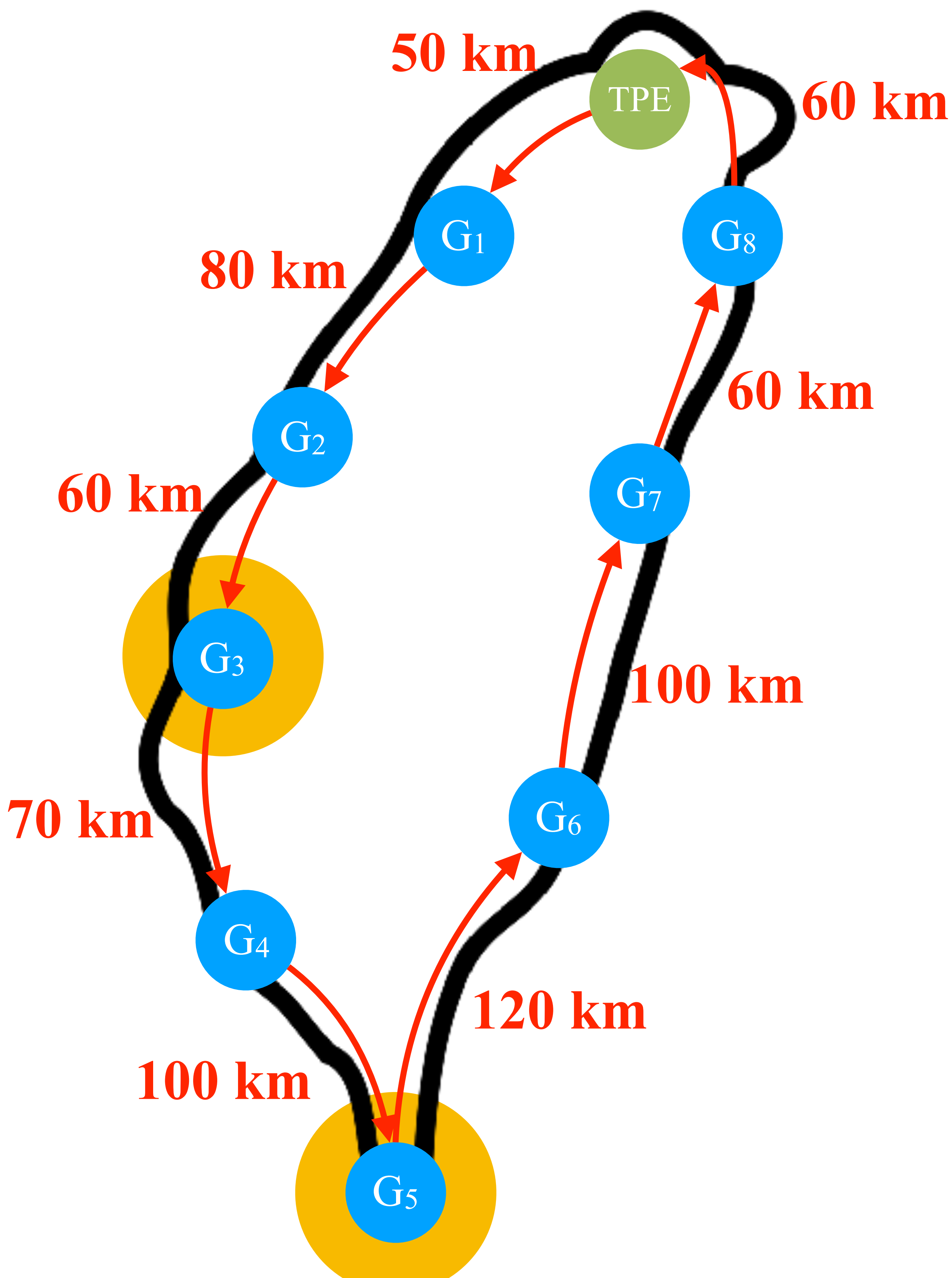


Alice has an old car. Once the car gets filled up, it can run for 200 km without any stop.

Alice would like to drive around Taiwan counterclockwise starting from Taipei. Her car is filled up in Taipei, and can be refilled at any Gi along the route.

Goal: minimize the number of stops to pump gas in the tour.

Pumping Gas Problem

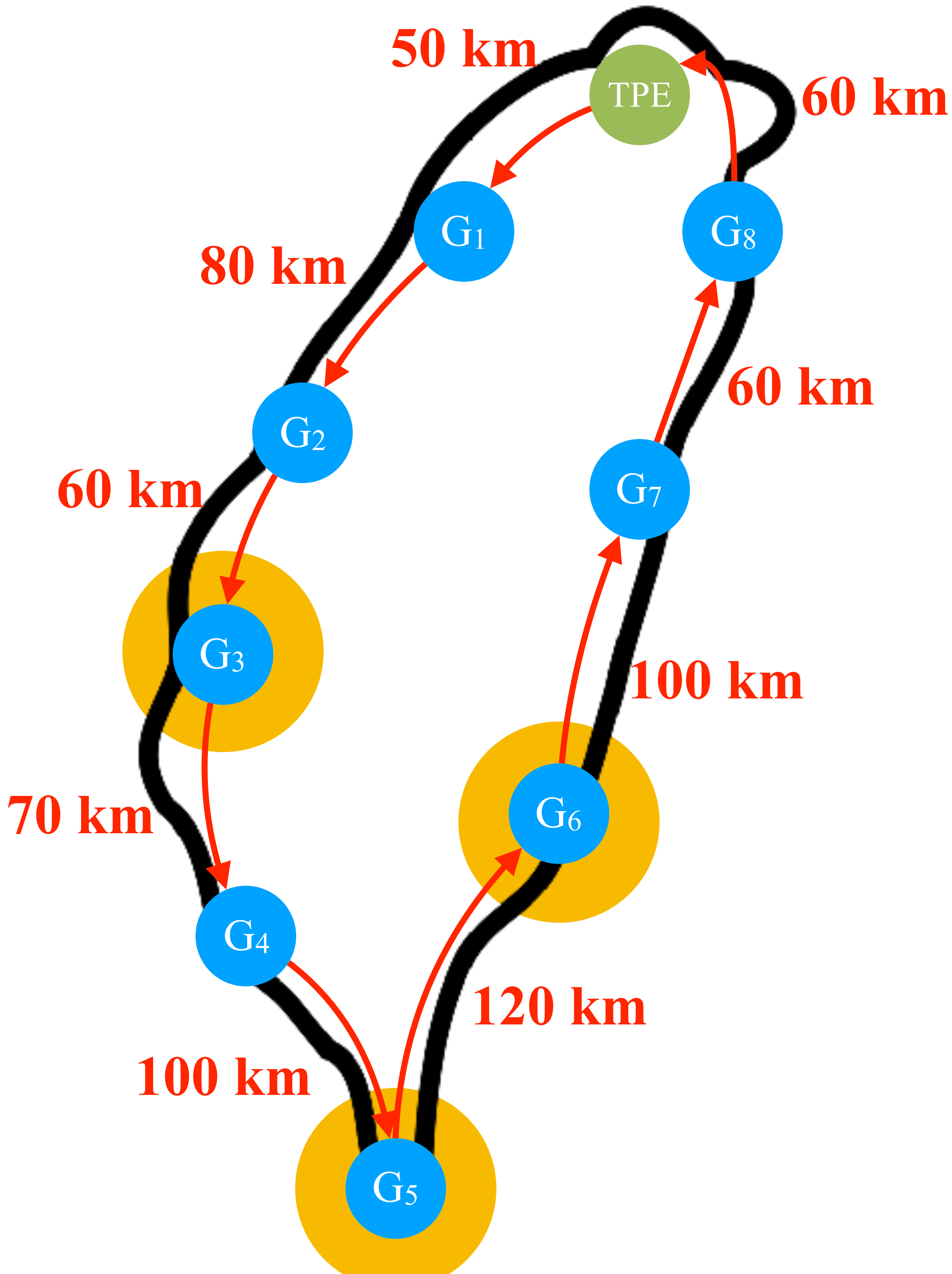


Alice has an old car. Once the car gets filled up, it can run for 200 km without any stop.

Alice would like to drive around Taiwan counterclockwise starting from Taipei. Her car is filled up in Taipei, and can be refilled at any Gi along the route.

Goal: minimize the number of stops to pump gas in the tour.

Pumping Gas Problem

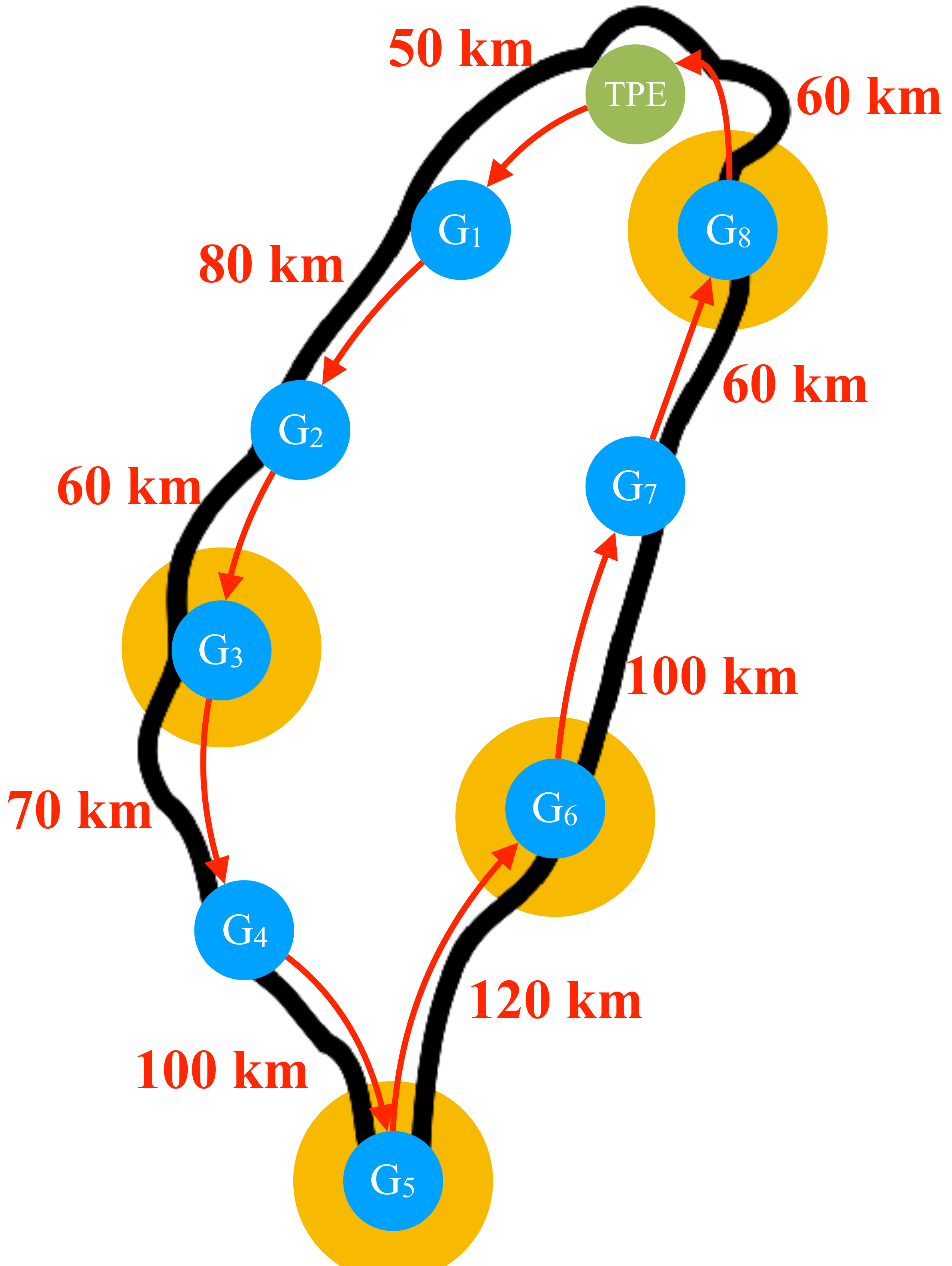


Alice has an old car. Once the car gets filled up, it can run for 200 km without any stop.

Alice would like to drive around Taiwan counterclockwise starting from Taipei. Her car is filled up in Taipei, and can be refilled at any Gi along the route.

Goal: minimize the number of stops to pump gas in the tour.

Pumping Gas Problem



Alice has an old car. Once the car gets filled up, it can run for 200 km without any stop.

Alice would like to drive around Taiwan counterclockwise starting from Taipei. Her car is filled up in Taipei, and can be refilled at any Gi along the route.

Goal: minimize the number of stops to pump gas in the tour.

Exercise

Give an O(n)-time algorithm to solve the pumping gas problem, and formally prove the correctness of your algorithm.

Knapsack Problems

Unweighted Knapsack Problem

Input: A set S of n stones where the i-th stone has

weight $w_i = 1$ and value $v_i \geq 0$.

Output: A subset T of S so that the total value of the stones in T is maximized and the total weight of the stones in T $\leq$ m.

Example.

What is T if m = 3?

Exercise

Give an O(n)-time algorithm to solve the unweighted knapsack problem.

Fractional Knapsack Problem

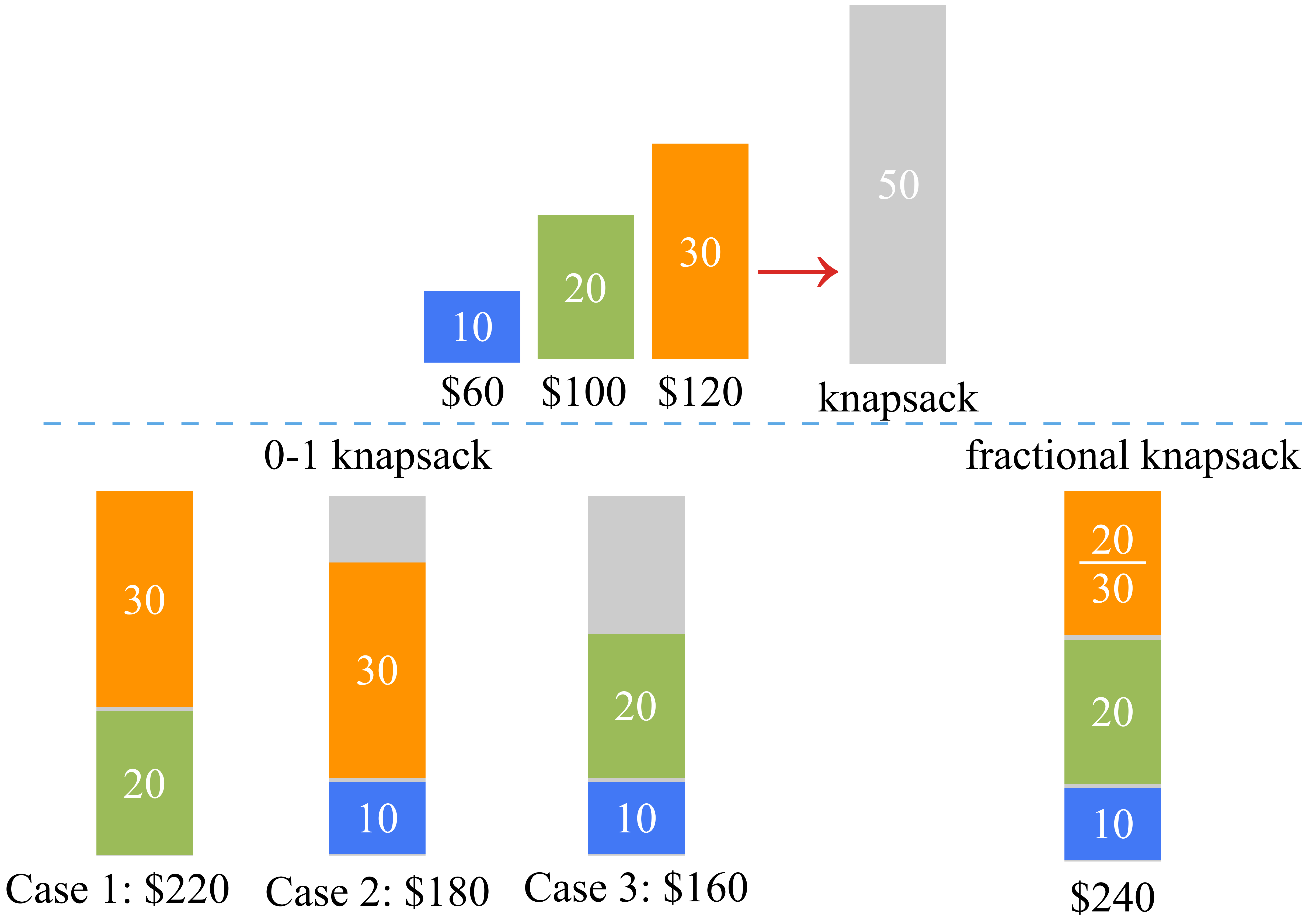
Input: A set S of n stones where the i-th stone has

weight $w_i \geq 0$ and value $v_i \geq 0$.

Output: for each stone s_i, output a fraction p_i indicating that a p_i portion of the stone is kept in the knapsack so that

(1) $\sum_{1 \leq i \leq n} p_i w_i \leq m$ and (2) $\sum_{1 \leq i \leq n} p_i v_i$ is maximized.

Fractional Knapsack Problem v.s. 0-1 Knapsack Problem



Exercise

Give an $O(n)$-time algorithm to solve the fractional knapsack problem.

Exercise

Assume that every stone has an integral weight in [0, m]. Give an O(nm)-time algorithm to solve the 0-1 knapsack problem by DP.

Hint. Let opt[i][j] be the best value to pack the first i stones into a knapsack of capacity j, so $opt[i][j] = \max(opt[i-1][j], opt[i-1][j-w_i] + v_i)$.

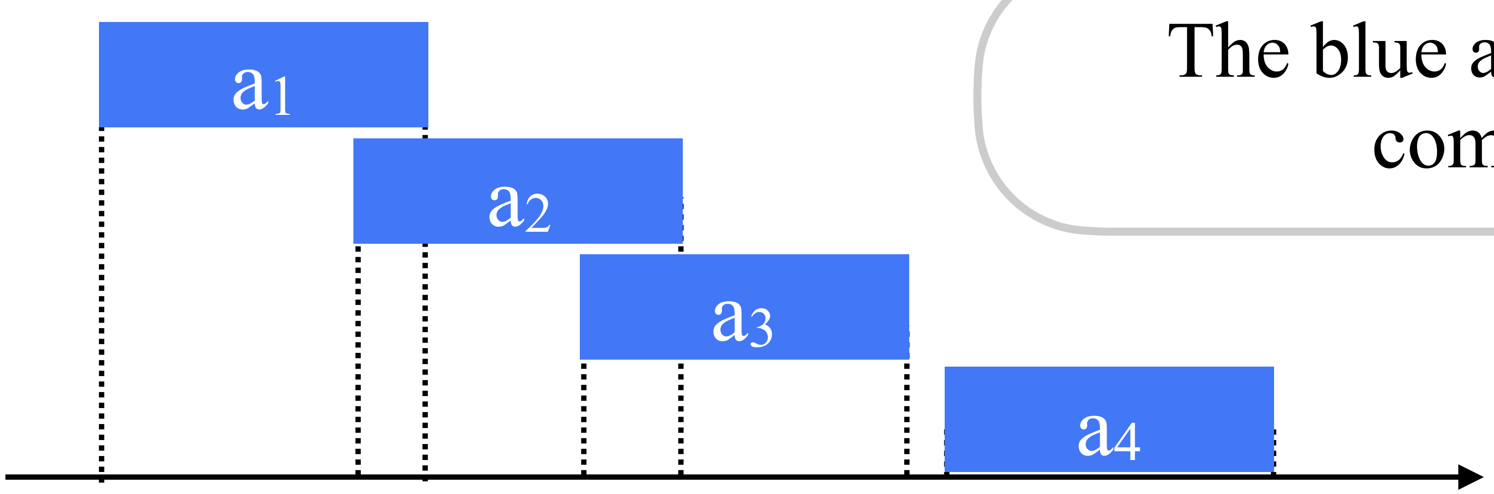
Activity Selection

Activity Selection Problem

Input: n activities a_1, a_2, …, a_n where each activitiy a_i has a starting time s_i and a finish time f_i. If a_i is selected, then a_i takes place during the time interval $[s_i, f_i)$ where $s_i < f_i$. We say two activities a_i, a_j are compatible if their time intervals have no intersection, i.e. $s_i \geq f_j$ or $s_j \geq f_i$.

Output: a maximum-size subset of mutually compatible activities.

--- Example ---



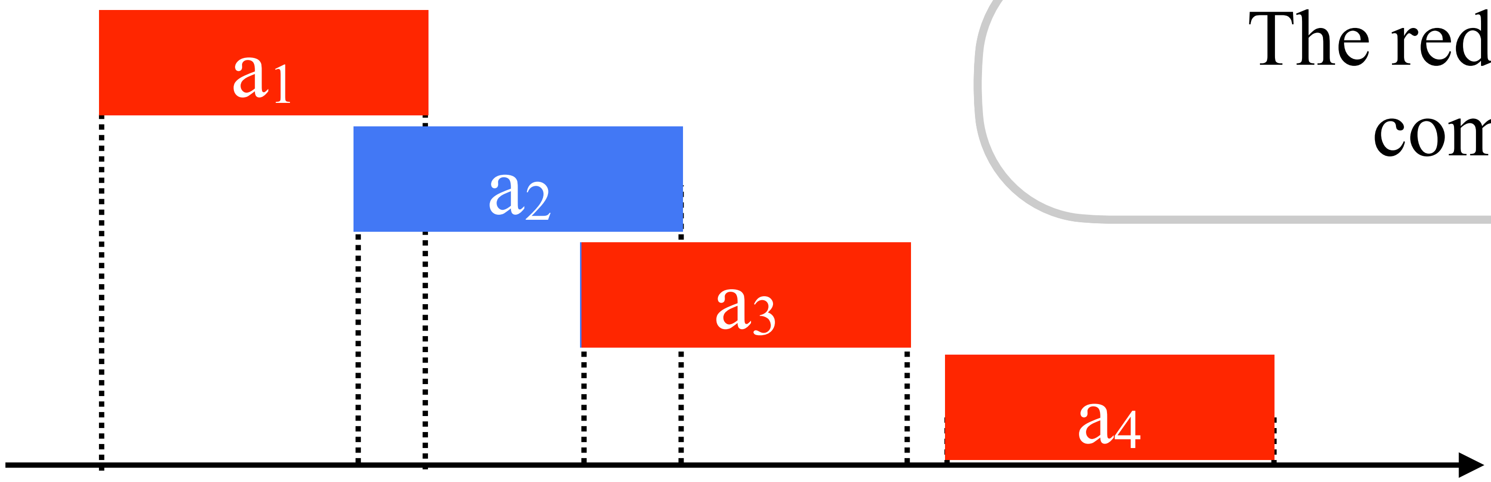
The blue activities are not compatitible.

Activity Selection Problem

Input: n activities $a_1, a_2, \ldots, a_n$ where each activitiy a_i has a starting time s_i and a finish time f_i. If a_i is selected, then a_i takes place during the time interval $[s_i, f_i)$ where $s_i < f_i$. We say two activities a_i, a_j are compatible if their time intervals have no intersection, i.e. $s_i \geq f_j$ or $s_j \geq f_i$.

Output: a maximum-size subset of mutually compatible activities.

--- Example ---



The red activities are compatitible.

Divide and Conquer

ASP(x, y){ // return the max-size subset of activities a_{x+1}, a_{x+2}, …, a_{y-1} so that they are mutually compatible and they are compatible with a_x and a_y, assuming that $f_x \leq f_{x+1} \leq \ldots \leq f_y$

    int opt = 0;

    for(i = x+1; i $<$ y; ++i){
        if (a_i is compatible with a_x and a_y){ // select a_i
            opt = (opt $<$ 1+ASP(i, y)) ? 1+ASP(i, y) : opt;
        }
    }

    return opt;
}

Divide and Conquer

ASP(x, y){ // return the max-size subset of activities a_{x+1}, a_{x+2}, …, a_{y-1} so that they are mutually compatible and they are compatible with a_x and a_y, assuming that $f_x \leq f_{x+1} \leq \ldots \leq f_y$

Why do we need this?

```
    int opt = 0;

    for(i = x+1; i < y; ++i){
        if (a_i is compatible with a_x and a_y){ // select a_i
            opt = (opt < 1+ASP(i, y)) ? 1+ASP(i, y) : opt;
        }
    }

    return opt;
}
```

The initial call is ASP(0, n+1) assuming that $s_0 = f_0 = -\infty$ and $s_{n+1}=f_{n+1} = \infty$.

Dynamic Programming

```
ASP(x, y, sol[][]){ // return the max-size subset of activities a_{x+1}, a_{x+2}, …, a_{y-1} so that they are mutually compatible and they are compatible with a_x and a_y, assuming that f_x ≤ f_{x+1} ≤ … ≤ f_y
    if(sol[x][y] ≥ 0) return sol[x][y];
    int opt = 0;

    for(i = x+1; i < y; ++i){
        if (a_i is compatible with a_x and a_y){ // select a_i
            opt = (opt < 1+ASP(i, y, sol)) ? 1+ASP(i, y, sol) : opt;
        }
    }

    return sol[x][y] = opt;
}
```

The initial call is ASP(0, n+1, sol = {-∞}).

Dynamic Programming

```
ASP(x, y, sol[][]){ // return the max-size subset of activities a_{x+1}, a_{x+2}, …,
a_{y-1} so that they are mutually compatible and they are compatible with a_x
and a_y, assuming that f_x ≤ f_{x+1} ≤ … ≤ f_y
    if(sol[x][y] ≥ 0) return sol[x][y];
    int opt = 0;

    for(i = x+1; i < y; ++i){
        if (a_i is compatible with a_x and a_y){ // select a_i
            opt = (opt < 1+ASP(i, y, sol)) ? 1+ASP(i, y, sol) : opt;
        }
    }

    return sol[x][y] = opt;
}
```

The initial call is
ASP(0, n+1, sol = {-∞}).

The running time is $O(n^3)$
because there are $O(n^2)$ subproblems and
each needs $O(n)$ time.

Greedy Algorithm - Recursion

```
ASP(x, y, sol[][]){ // return the max-size subset of activities a_{x+1}, a_{x+2}, …,
a_{y-1} so that they are mutually compatible and they are compatible with a_x
and a_y, assuming that f_x ≤ f_{x+1} ≤ … ≤ f_y
    if(sol[x][y] ≥ 0) return sol[x][y];
    int opt = 0;

    for(i = x+1; i < y; ++i){
        if (a_i is compatible with a_x and a_y){ // select a_i
            opt = (opt < 1+ASP(i, y, sol)) ? 1+ASP(i, y, sol) : opt;
            break; // a greedy choice: once a compatible a_i is found, add it
                       into the optimal solution no matter what
        }
    }

    return sol[x][y] = opt;
}
```

The initial call is ASP(0, n+1, sol = {-∞}).

Greedy Algorithm - Recursion

```
ASP(x, y, sol[][]){ // return the max-size subset of activities a_{x+1}, a_{x+2}, …,
a_{y-1} so that they are mutually compatible and they are compatible with a_x
and a_y, assuming that f_x ≤ f_{x+1} ≤ … ≤ f_y
    if(sol[x][y] ≥ 0) return sol[x][y];
    int opt = 0;

    for(i = x+1; i < y; ++i){
        if (a_i is compatible with a_x and a_y){ // select a_i
            opt = (opt < 1+ASP(i, y, sol)) ? 1+ASP(i, y, sol) : opt;
            break; // a greedy choice: once a compatible a_i is found, add it
                      into the optimal solution no matter what
        }
    }

    return sol[x][y] = opt;
}
```

The initial call is ASP(0, n+1, sol = {-∞}).

The running time is O(n). Why?

Why can we make the greedy choice?

```
for(i = x+1; i < y; ++i){
    if (a_i is compatible with a_x and a_y){ // select a_i
        opt = (opt < 1+ASP(i, y, sol)) ? 1+ASP(i, y, sol) : opt;
        break; // a greedy choice: once a compatible a_i is found, add it
                  into the optimal solution no matter what
    }
}
```

Assume that the optimum solution is

a_p	a_q	a_r	a_s	…	a_z

If $p = i$, then the greedy choice is on the way to find the optimal solution.
If $p < i$, why does the greedy choice ignore a_p, an earlier compatible one?
If $p > i$, then replacing a_p with a_i yield another optimal solution.

Summary

Here is a common technique for the algorithm design.

(1) Imagine what the optimum solution is.
(2) Give an initial guess.
(3) If the guess ≠ the optimum solution, find a way to morph the guess to the solution.

G. W. Busch

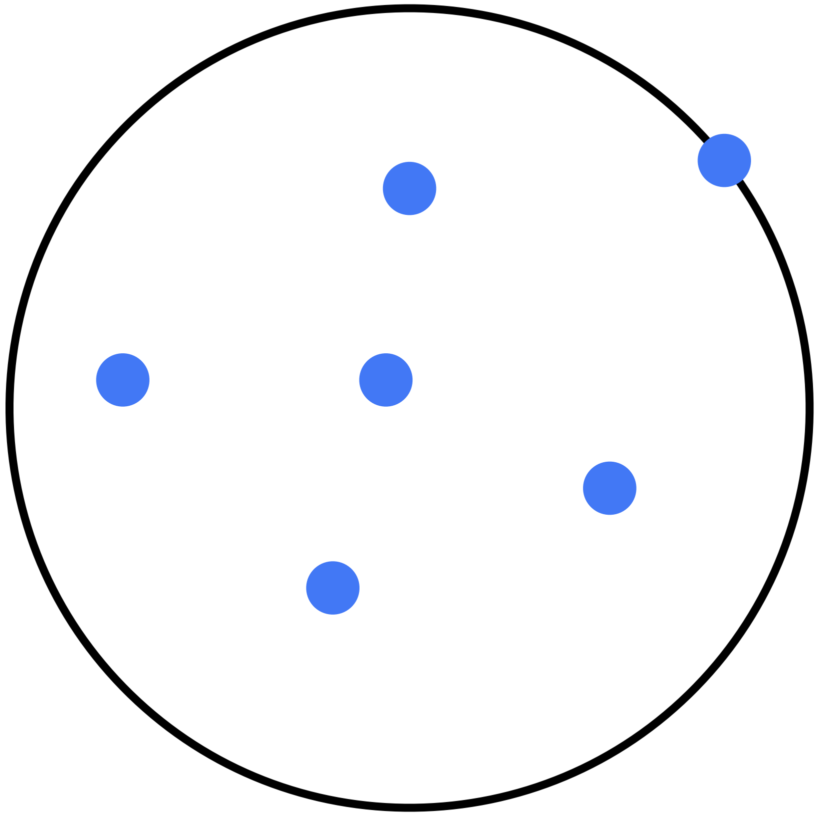
A. Schwarzenegger

Photo Credit: wikipedia.

Exercise

The Minimum Enclosing Ball Problem.

Given n points on a 2D plane, find the smallest circle so that each of the n point is either on the boundary of the circle or in its interior.



An enclosing ball

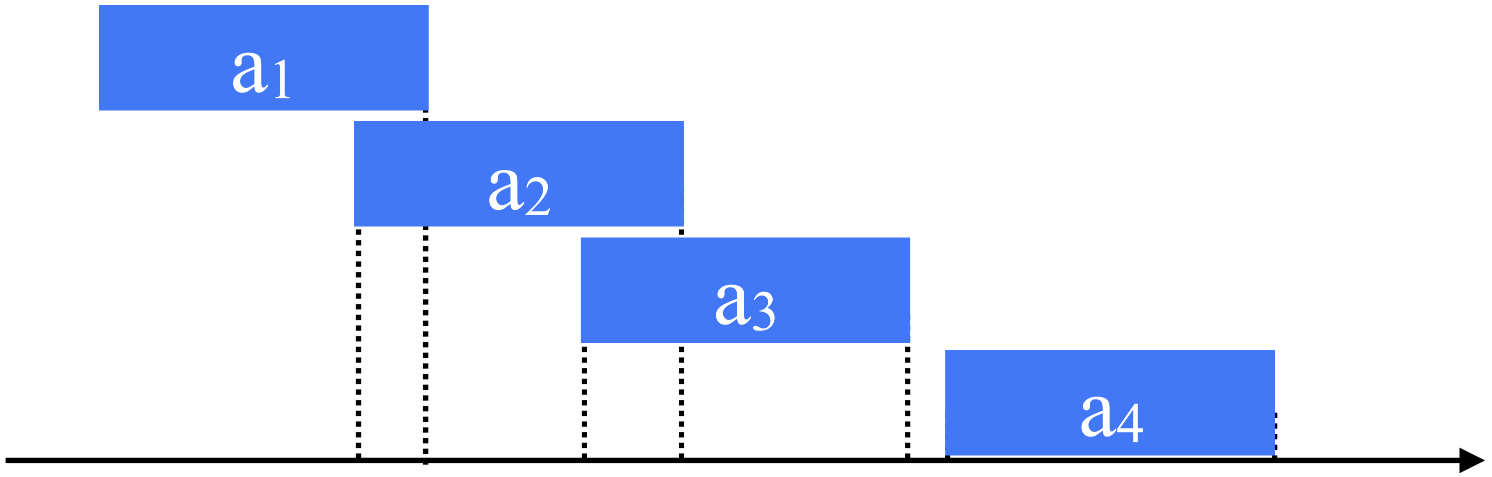
Greedy Algorithm - Loop Iterations

Sort a_i's by their finish time. Let $(s_0, f_0) = (-\infty, -\infty)$.

```
prev = 0;
count = 0;

for(i = 1; i ≤ n; ++i){
   if(s_i ≥ f_prev){
       prev = i;
       ++ count;
   }
}

return count;
```



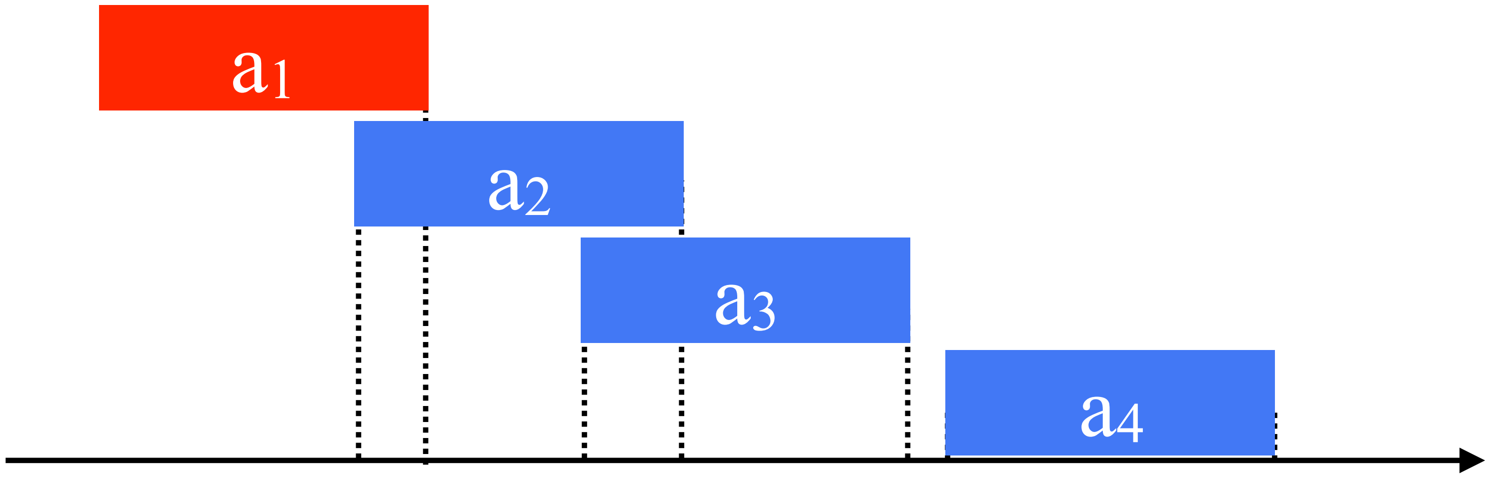
Greedy Algorithm - Loop Iterations

Sort a_i's by their finish time. Let $(s_0, f_0) = (-\infty, -\infty)$.

```
prev = 0;
count = 0;

for(i = 1; i ≤ n; ++i){
   if(s_i ≥ f_prev){
       prev = i;
       ++ count;
   }
}

return count;
```



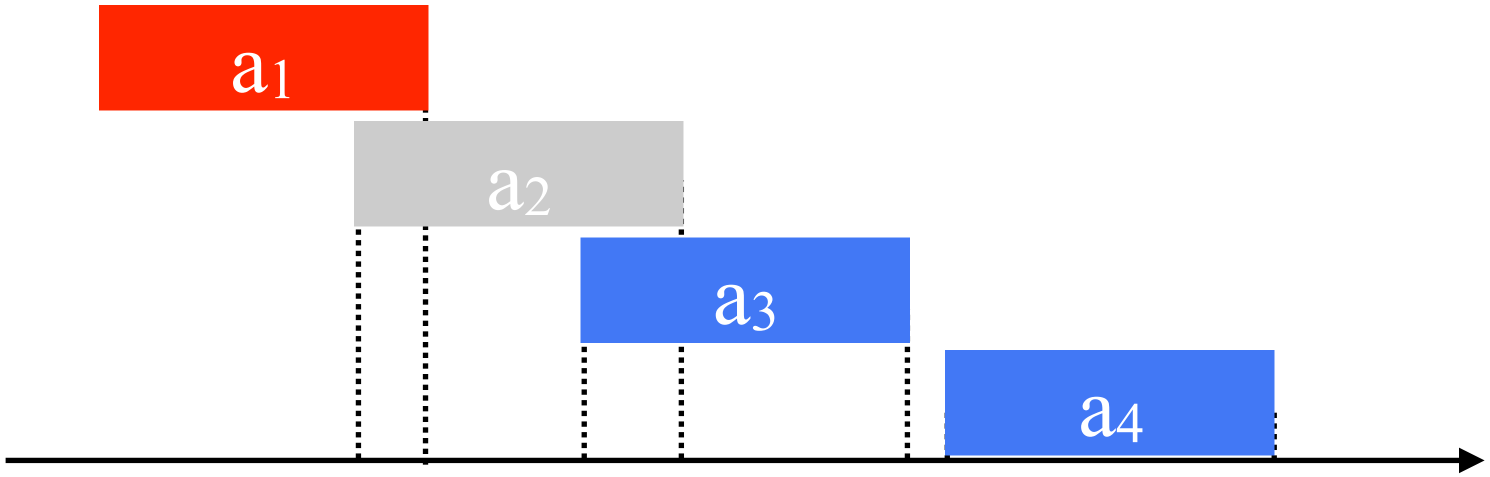
Greedy Algorithm - Loop Iterations

Sort a_i's by their finish time. Let $(s_0, f_0) = (-\infty, -\infty)$.

```
prev = 0;
count = 0;

for(i = 1; i ≤ n; ++i){
   if(s_i ≥ f_prev){
       prev = i;
       ++ count;
   }
}

return count;
```



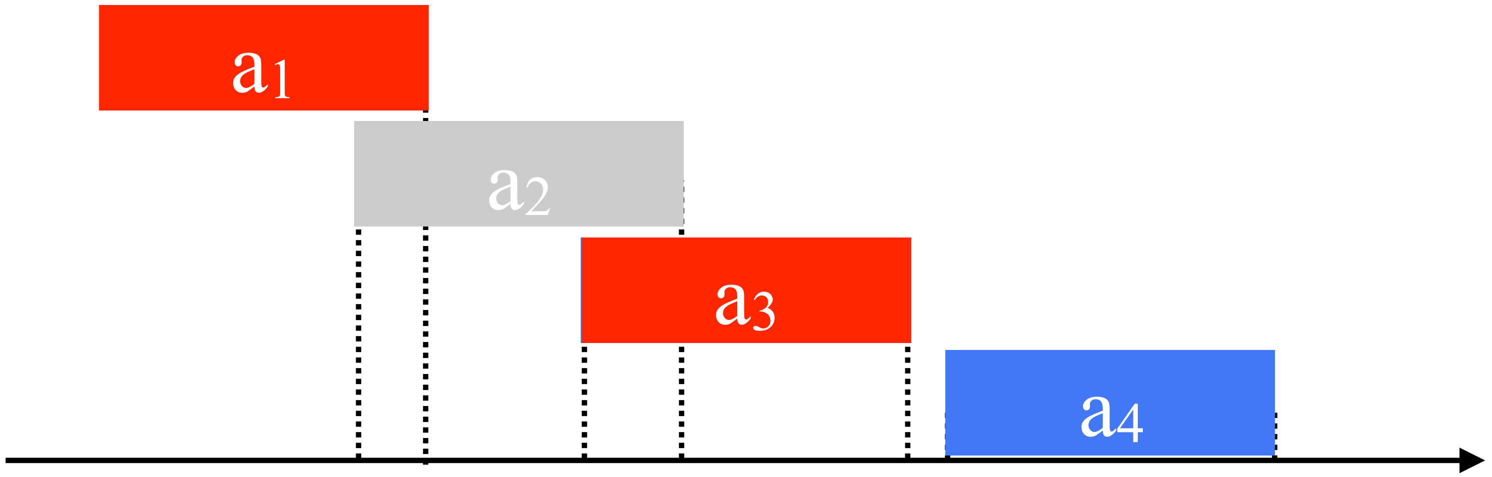
Greedy Algorithm - Loop Iterations

Sort a_i's by their finish time. Let $(s_0, f_0) = (-\infty, -\infty)$.

```
prev = 0;
count = 0;

for(i = 1; i ≤ n; ++i){
   if(s_i ≥ f_prev){
       prev = i;
       ++ count;
   }
}

return count;
```



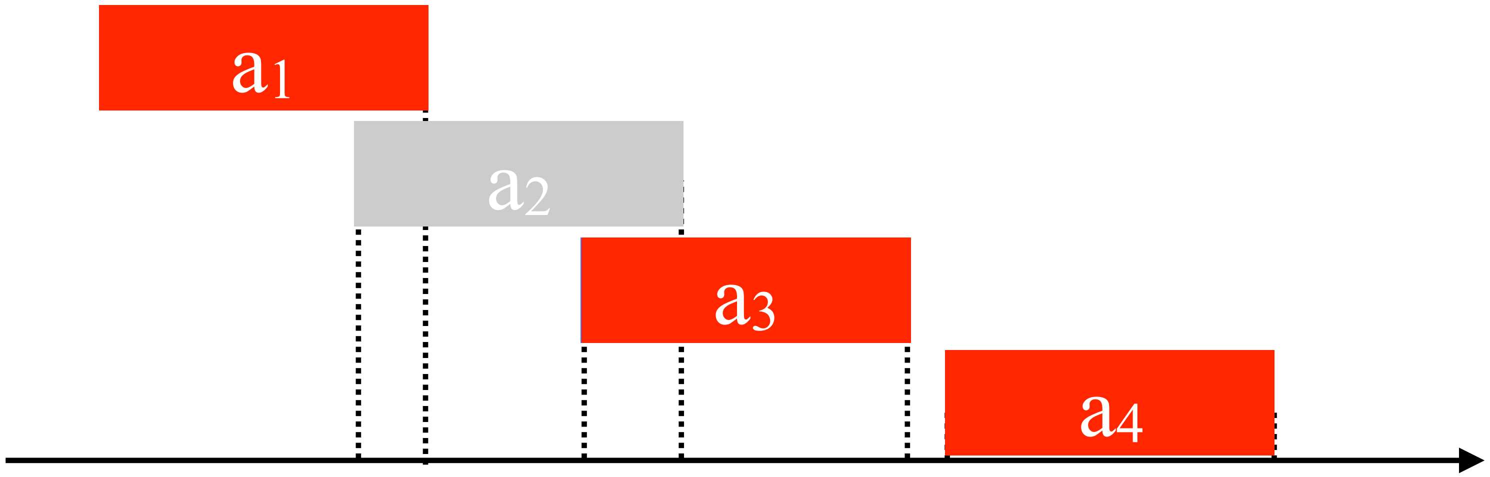
Greedy Algorithm - Loop Iterations

Sort a_i's by their finish time. Let $(s_0, f_0) = (-\infty, -\infty)$.

```
prev = 0;
count = 0;

for(i = 1; i ≤ n; ++i){
   if(s_i ≥ f_prev){
       prev = i;
       ++ count;
   }
}

return count;
```



Exercise

Prove that the activity selection problem has a lower bound of $\Omega(n \log n)$ in the comparison-based model.

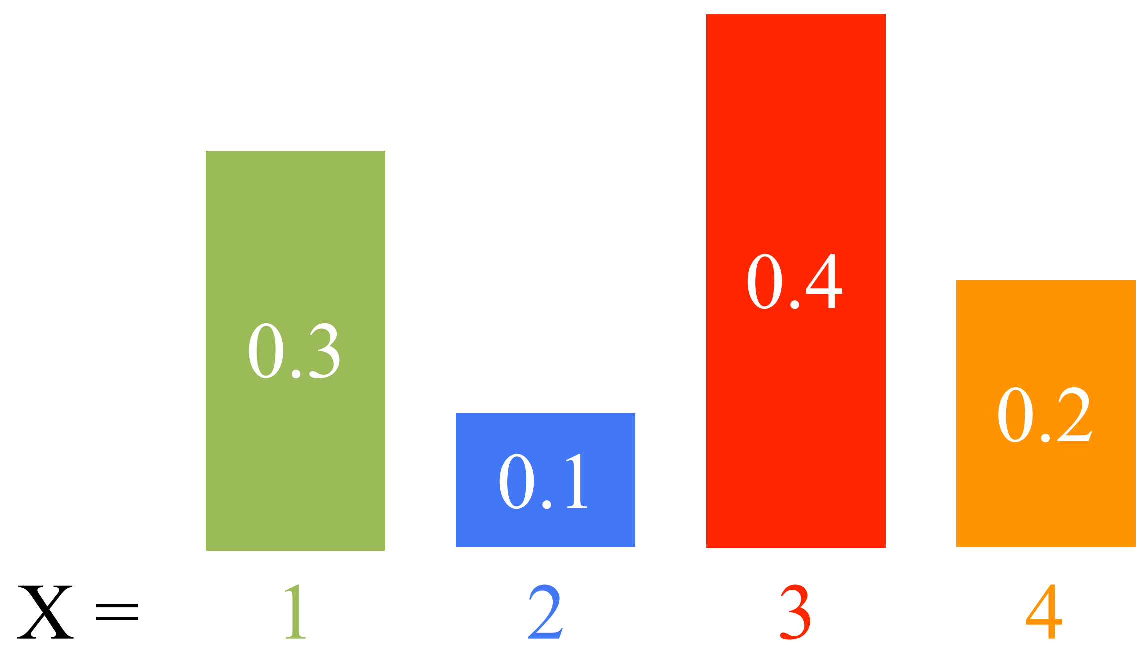
(Hint. Element uniqueness problem.)

Gategorical Sampling

Categorical Sampling

Given a categorical distribution P where a random variate X sampled from P follows the distribution $\Pr[X = i] = p_i$ for each i in [1, n].

Implement a function F (allow preprocessing). Every time F is invoked, it outputs a random variate X that follows P.



An O(log n)-time Approach

Sample a uniform random number X_U from [0, 1).

```
for(i = 1; i ≤ n; ++i){
   if(X_U < p_1 + p_2 + … + p_i){
       return i;
   }
}
```

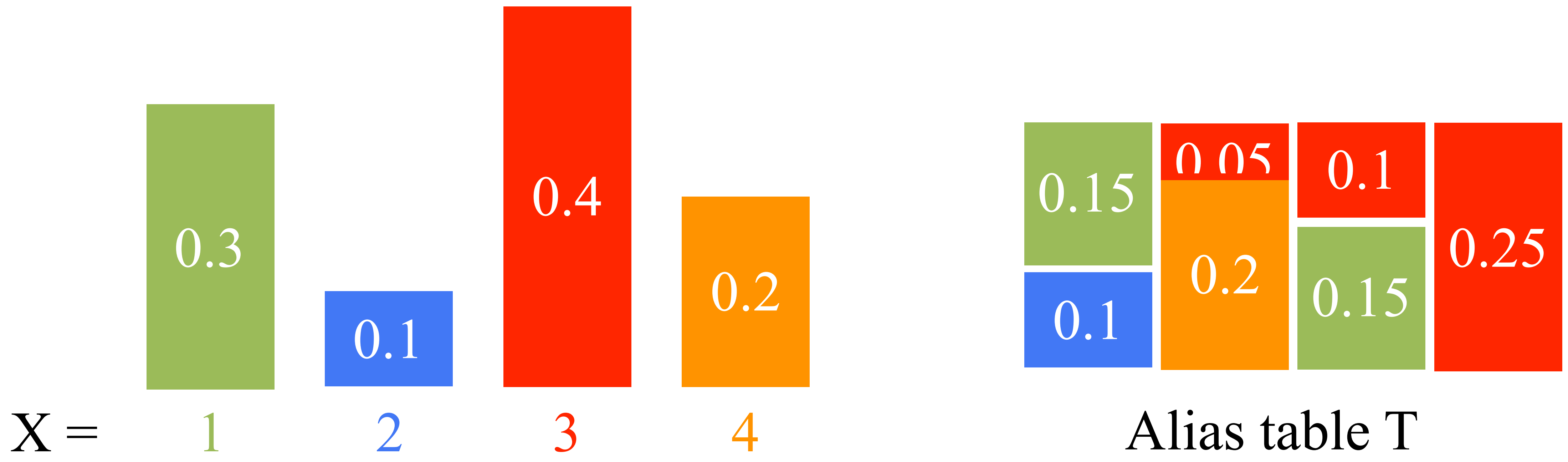
--- Details ---

In O(n) time, one can build an array A of prefix sum. That is,

$$A[i] = \sum_{k \leq i} p_k.$$

Given A, every sample can be drawn by sampling an X_U and binary search where X_U is over A.

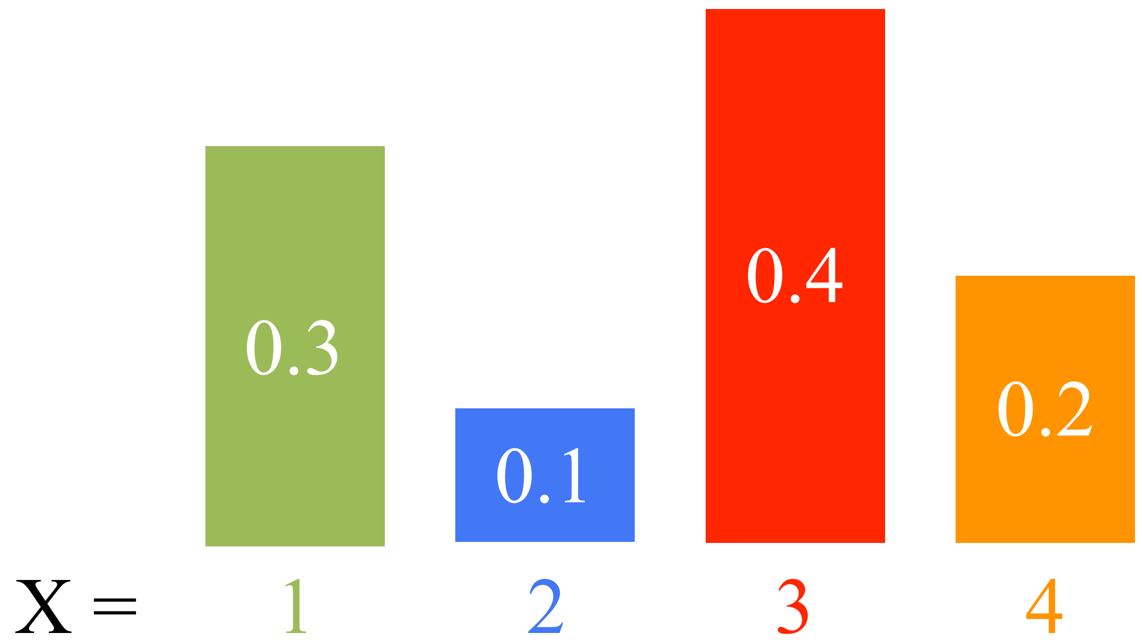
An O(1)-time Approach



Build an alias table T of n entries so that

(1) each entry contains portions from at most 2 categories.
(2) the portions in each entry sum to 1/n.

An O(1)-time Approach



Alias table T

Sample a uniform random number X_U from [0, 1).

Let k = ceil(X_U/(1/n)). Let A, B be the portions in T[k].

Sample another uniform random number Y_U from [0, 1/n).

return A if |A| < Y_U or otherwise B.

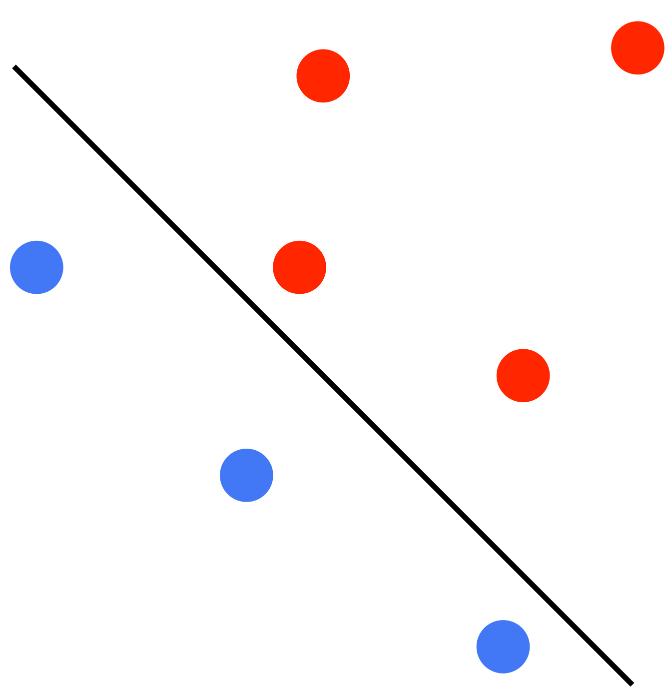
Exercise

Give an O(n)-time algorithm to build the alias table of $p_1, p_2, \ldots, p_n$.

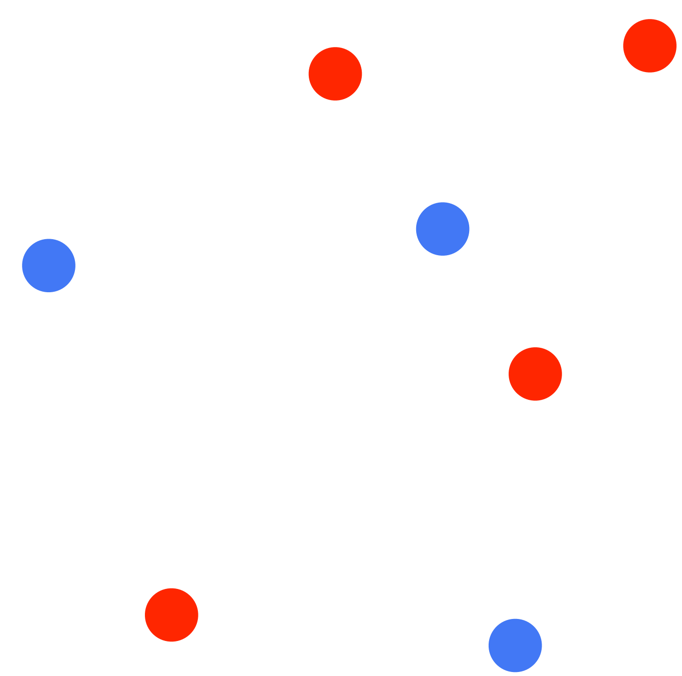
Hint. Let $p_k < 1/n$ for some k and $p_\ell > 1/n$ for some ℓ. Fill up the current entry in T with all of p_k and $(1/n-p_k)$ portion of p_ℓ.

Exercise

Given n points on a 2D plane. The color of each point is blue or red. Decide whether there exists a line that seperates the points into two halves so that each half contains points of same color.



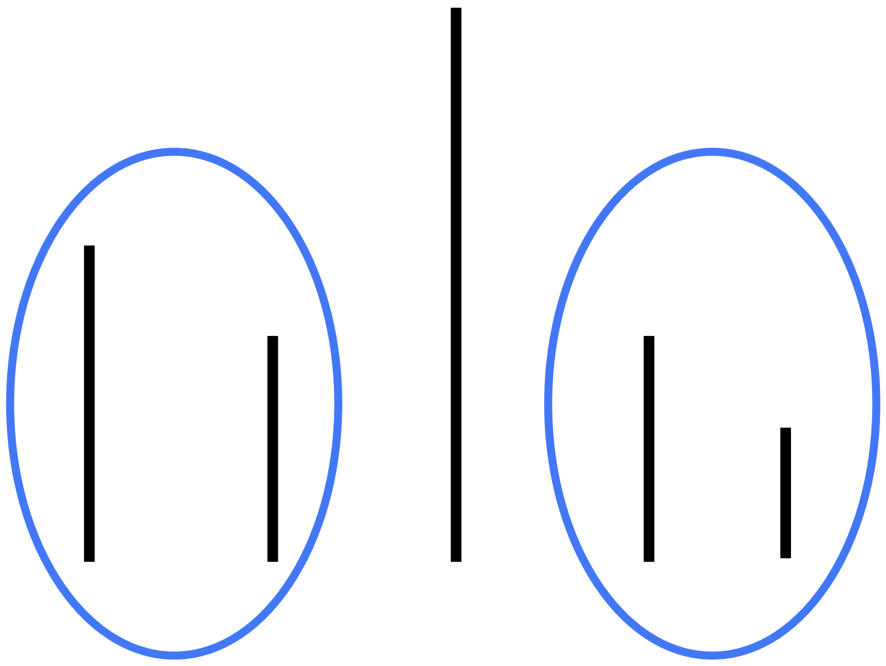
Yes



No

Exercise

Given n chopsticks that have length $\ell_1, \ell_2, \ldots, \ell_n \geq 0$. If two chopsticks have length difference smaller than d, then we can pair the two chopsticks. Maximize the number of paired chopsticks, noting that each chopstick can join at most 1 pair.



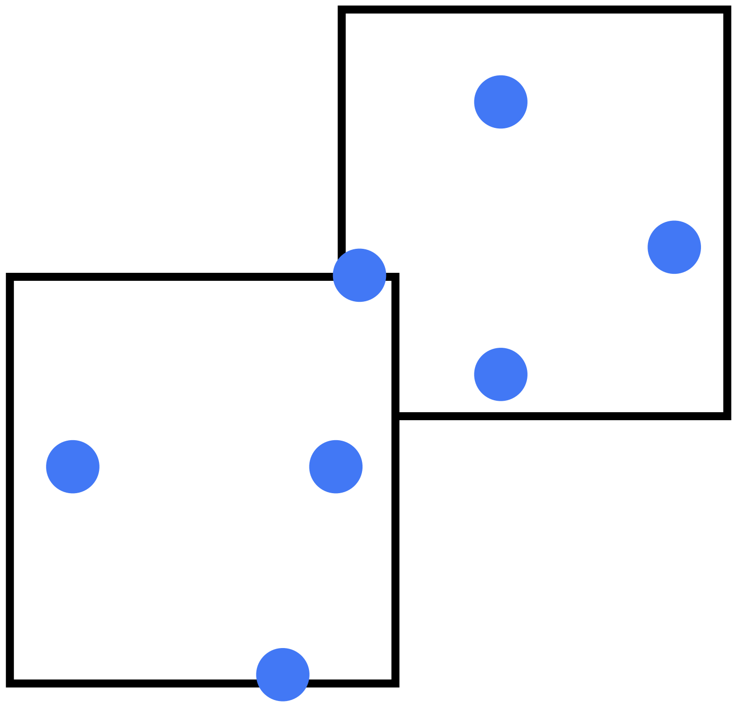
Another Common Technique

(1) Let some parameter be a fixed value and see whether you can solve the problem.

(2) Observe the relationship (e.g. monotonicity) between the solutions for two different fixed values.

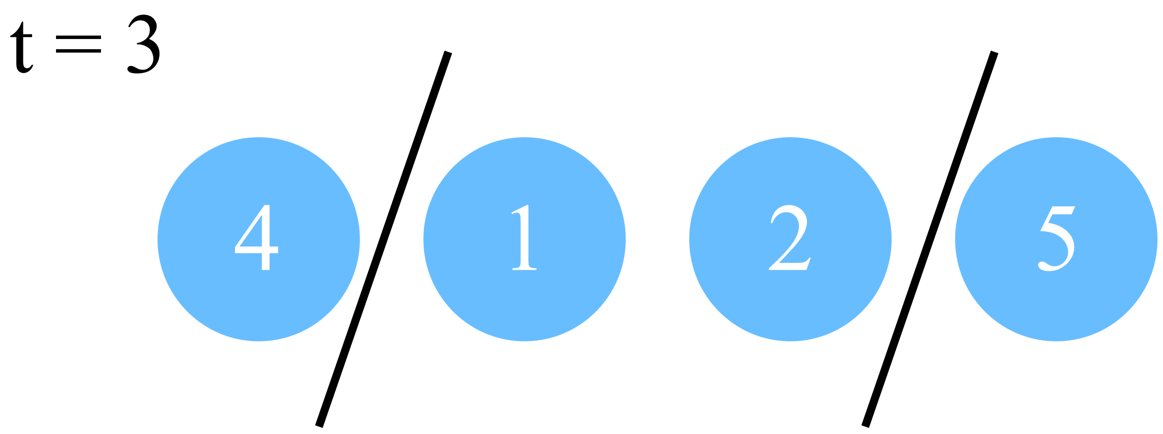
Exercise

Given n points on a 2D plane, cover all the points by two squares of length L so that L is minimized. These two squares may overlap.



Exercise

Given a sequence of n positive integers $a_1, a_2, ..., a_n$. Define weight of a consecutive subsequence to be the sum of elements in it. Partition these n integers into t consecutive subsequences so that the maximum weight of the t subsequences is minimized.



Exercise

Given n chopsticks that have integral length $\ell_1, \ell_2, ..., \ell_n \geq 0$. Find k pairs of chopsticks from the n given ones so that the maximum length difference in a pair is minimized.