Отчет по лабораторной работе 4

Неполнодоступная двухсервисная модель Эрланга с резервированием для заявок второго типа и разными интенсивностями обслуживания. Первыми заполняются зарезервированные приборы.

Генералов Даниил, 1032212280

Рассматривается неполнодоступная двухсервисная модель Эрланга с разными интенсивностями обслуживания и резервированием для обслуживания запросов на предоставление услуги 2-го типа. Запросы на предоставление услуги 2-го типа сначала заполняют зарезервированную емкость.

- C -- общее число приборов;
- g -- число полнодоступных приборов;
- $C - g$ -- число зарезервированных приборов;
- λ_1, λ_2 -- интенсивность поступления запросов 1, 2-го типа;
- μ^{-1} -- среднее время обслуживания запроса 1, 2-го типа;
- ρ_1, ρ_2 -- интенсивность предложенной нагрузки, создаваемой запросами 1, 2-го типа;
- $X(t)$ -- число запросов, обслуживаемых в системе в момент времени t, $t \geq 0$;
- X -- пространство состояний системы;
- n -- число обслуживаемых в системе запросов;
- B_1, B_2 -- множество блокировок запросов 1, 2-го типа;
- S_1, S_2 -- множество приема запросов 1, 2-го типа.

1. подключение библиотек, определение функций

Для расчета больших факториалов нам потребуется длинная арифметика, а для рисования графиков -- библиотека для визуализации данных.

In [3]:
```
:dep num = { version = "^0.4.3" }
:dep plotters = { version = "^0.3.6", default-features = false, features = [

extern crate num;
use num::BigRational as R;
use num::BigInt as I;
use num::BigUint as U;
```

```
use num::Integer;
use num::traits::ConstZero;
use num::FromPrimitive;
use num::ToPrimitive;

extern crate plotters;
use plotters::prelude::*;
```

Для удобства конвертации стандартных чисел в числа длинной арифметики используются helper-функции.

In [4]:
```
fn u(i: usize) -> U {
    U::from_usize(i).unwrap()
}

fn rr(i: f64) -> R {
    R::from_float(i).unwrap()
}
```

Для вычисления факториала нет стандартной функции, и очевидные подходы не работают с длинной арифметикой, поэтому эта функция считает это за нас.

In [5]:
```
fn factorial(n: &U) -> R {
    let mut c = n.clone();
    let one = I::from_i8(1).unwrap();
    let mut out = R::new(one.clone(), one.clone());
    while c > U::ZERO {
        out *= R::new(I::from_biguint(num::bigint::Sign::Plus, c.clone()), o
        c -= 1u32;
    }
    out
}
```

Эта функция возвращает список, состоящий из только уникальных значений в указанном списке. Это нужно для того, чтобы упростить вычисление набора состояний.

In [6]:
```
fn make_unique<T: Eq + std::hash::Hash>(v: Vec<T>) -> Vec<T> {
    let mut set = std::collections::HashSet::new();
    for x in v {
        set.insert(x);
    }
    set.into_iter().collect()
}
```

Эта функция отображает график функции, принимая на вход список X-Y пар.

In [7]:
```
fn draw_chart(data: &Vec<(f32, f32)>, name: impl ToString) -> plotters::evcx
    let minx = data.iter().min_by(|a, b| a.0.partial_cmp(&b.0).unwrap_or(std
    let maxx = data.iter().max_by(|a, b| a.0.partial_cmp(&b.0).unwrap_or(std
    let miny = data.iter().min_by(|a, b| a.1.partial_cmp(&b.1).unwrap_or(std
    let maxy = data.iter().max_by(|a, b| a.1.partial_cmp(&b.1).unwrap_or(std
```

```
    let figure = evcxr_figure((640, 480), |root| {
        root.fill(&WHITE)?;
        let mut chart = ChartBuilder::on(&root)
            .caption(name.to_string(), ("Arial", 50).into_font())
            .margin(5)
            .x_label_area_size(30)
            .y_label_area_size(30)
            .build_cartesian_2d(minx..maxx, miny..maxy)?;

        chart.configure_mesh().draw()?;

        chart.draw_series(LineSeries::new(
            data.clone(),
            &RED,
        )).unwrap();

        // chart.configure_series_labels()
        //     .background_style(&WHITE.mix(0.8))
        //     .border_style(&BLACK)
        //     .draw()?;
        Ok(())
    });
    return figure;
}
```

2. входные параметры

Здесь задаются параметры, которые определяют модель. Чтобы попробовать запустить вычисления с другими значениями, вы можете поменять эту ячейку и перезапустить ее и все ячейки ниже.

При разработке я использовал следующие значения для теста:

- $C = 4$
- $g = 3$
- $\rho_1 = \frac{1}{2}$
- $\rho_2 = 1$
- $\mu = 10$

In [8]:
```
let C = 4;
let g = 3;
let rho_1 = 0.5;
let rho_2 = 1.0;

let mu1 = 10.0;
```

3. вычисления

Для того, чтобы определить набор состояний, мы рассматриваем все возможные пары $(x, y) \geq (0, 0)$, где $x + y \leq C$ и $n_1 \leq g$. Этот цикл находит их и сохраняет в список.

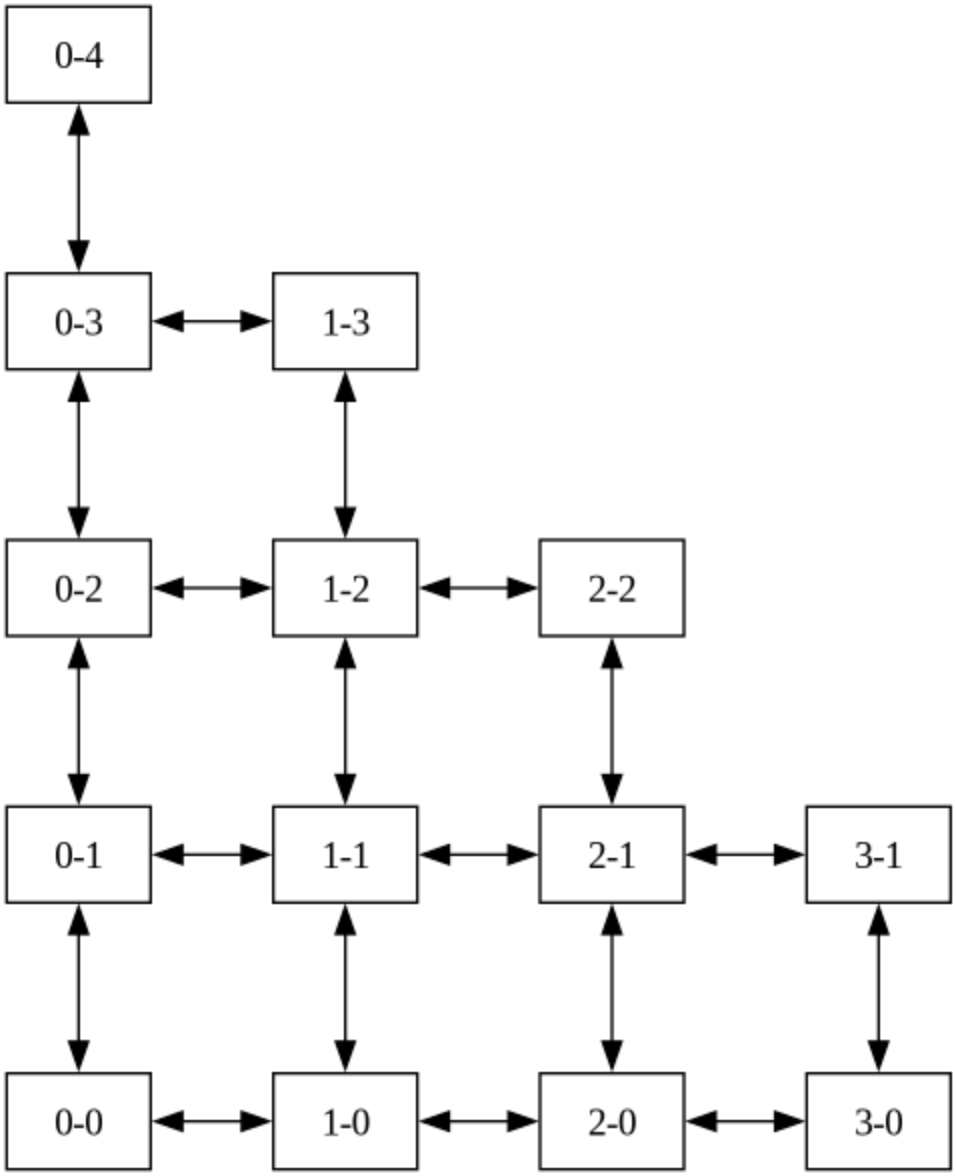
In [9]:
```
println!("X = (n1, n2): n1 = 0,{g}, n2=0,{C}, n1+n2 <= {C}");
let mut states = vec![];
for n1 in 0..=g {
    for n2 in 0..=C {
        if n1 + n2 <= C {
            states.push((n1, n2));
        }
    }
}
println!("|X| = {}", states.len());
```

```
X = (n1, n2): n1 = 0,3, n2=0,4, n1+n2 <= 4
|X| = 14
```

Для того, чтобы показать набор состояний, мы должны нарисовать граф, который содержит все состояния в наборе X, и имеет соединения между соседними ячейками. Этот код формирует такой граф, затем визуализирует его через Graphviz, и затем показывает в виде картинки.

In [10]:
```
:dep graphviz-rust = { version = "^0.9.0" }
:dep base64 = { version = "^0.22.1" }
use base64::prelude::*;

let mut dot_graph = String::new();

dot_graph.push_str("graph G {\n");
dot_graph.push_str("  rankdir=LR;\n");
dot_graph.push_str("  layout=nop;\n");
dot_graph.push_str("  node [shape=box];\n");

for (n1, n2) in states.iter() {
    dot_graph.push_str(&format!("S{}_{} [label=\"{}-{}\", pos=\"{:.1}, {:.1}
    let (n1, n2) = (*n1, *n2);
    if states.contains(&(n1+1, n2)) {
        dot_graph.push_str(&format!("S{}_{} -- S{}_{} [dir=both]\n", n1, n2,
    }
    if states.contains(&(n1, n2+1)) {
        dot_graph.push_str(&format!("S{}_{} -- S{}_{} [dir=both]\n", n1, n2,
    }
}

dot_graph.push_str("}\n");


let png = graphviz_rust::exec_dot(dot_graph, vec![graphviz_rust::cmd::Format
let png_base64 = BASE64_STANDARD.encode(&png);
println!("EVCXR_BEGIN_CONTENT image/png\n{}\nEVCXR_END_CONTENT", png_base64)
```

Out[10]:



Состояния блокировки заявок -- это те, находясь в которых, система не может принять больше заявок первого или второго типов. Для обоих типов заявок таковыми являются те, когда общее количество заявок равно C (тогда даже приоритетные заявки нельзя обработать), но для заявок первого типа -- есть еще ситуация, когда заняты все выделенные слоты (когда количество заявок $n_1 = g$).

In [11]:

```
// boundary states
let mut b1 = vec![];
let mut b2 = vec![];
for (n1, n2) in states.iter().cloned() {
    if n1 == g {
        b1.push((n1, n2));
    }
    if n1 + n2 == C {
        b1.push((n1, n2));
        b2.push((n1, n2));
    }
}
```

```
let b1 = make_unique(b1);
let b2 = make_unique(b2);
println!("B1 = (n1, n2) \\in X: n1 = {g} \\lor n1 + n2 = {C} = {b1:?}");
println!("B2 = (n1, n2) \\in X: n1 + n2 = {C} = {b2:?}");
```

```
B1 = (n1, n2) \in X: n1 = 3 \lor n1 + n2 = 4 = [(3, 1), (3, 0), (0, 4), (1,
3), (2, 2)]
B2 = (n1, n2) \in X: n1 + n2 = 4 = [(1, 3), (0, 4), (2, 2), (3, 1)]
```

Множество приема запросов -- это обратное множество к множеству блокировки: $S_i = X \setminus B_i$. Здесь мы вычисляем его именно так, но также есть способ написать обратную формулу к формуле определения множества блокировки.

In [12]:
```
// interior states
let mut s1 = vec![];
let mut s2 = vec![];
for s in states.iter() {
    if !b1.contains(s) {
        s1.push(*s);
    }
    if !b2.contains(s) {
        s2.push(*s);
    }
}

println!("S1 = (n1, n2) \\in X: n1 < {} \\lor n1+n2 < {} = {s1:?}", g-1, C-1
println!("S2 = (n1, n2) \\in X: n1+n2 < {} = {s2:?}", C-1);
```

```
S1 = (n1, n2) \in X: n1 < 2 \lor n1+n2 < 3 = [(0, 0), (0, 1), (0, 2), (0, 3),
(1, 0), (1, 1), (1, 2), (2, 0), (2, 1)]
S2 = (n1, n2) \in X: n1+n2 < 3 = [(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1,
1), (1, 2), (2, 0), (2, 1), (3, 0)]
```

Для неполнодоступной модели Эрланга с двумя типами заявок известно, что любой замкнутый цикл имеет одинаковое произведение интенсивностей переходов в обе стороны, поэтому по критерию Колмогорова существует частичный баланс, и поэтому можно посчитать распределение вероятностей.

Для этого сначала расчитаем плотность вероятностей в каждой ячейке как пропорцию от p_{00}, а затем, учитывая, что мы имеем дело с распределением вероятностей, -- определим значение p_{00} относительно него.

In [13]:
```
use std::collections::HashMap;

fn make_probs(log: bool, rho_1: &f64, rho_2: &f64, states: &Vec<(i32, i32)>)

    let mut probs = HashMap::new();
    for (n1, n2) in states.iter() {
        let n1 = *n1;
        let n2 = *n2;
        if (n1, n2) != (0, 0) {
            let p = (rr(*rho_1 as f64).pow(n1) / factorial(&u(n1 as usize)))
            probs.insert((n1, n2), p);
        }
```

```
    }
    if log {
        println!("probs before normalization: ");
        for prob in probs.iter() {
            println!("{prob:?}");
        }
    }

    probs.insert((0, 0), R::from_i8(1).unwrap());

    let p00 = probs.iter().map(|(_, v)| v).sum::<R>().pow(-1);

    if log {
        println!("p00 = {} = {}", p00, p00.to_f64().unwrap());
    }

    probs.iter_mut().for_each(|(_, v)| *v *= &p00);

    probs
}

let probs = make_probs(true, &rho_1, &rho_2, &states);
println!("probs after normalization: ");
for prob in probs.iter() {
    println!("{prob:?}");
}
```

```
probs before normalization:
((1, 2), Ratio { numer: 1, denom: 4 })
((2, 0), Ratio { numer: 1, denom: 8 })
((0, 2), Ratio { numer: 1, denom: 2 })
((3, 0), Ratio { numer: 1, denom: 48 })
((2, 2), Ratio { numer: 1, denom: 16 })
((3, 1), Ratio { numer: 1, denom: 48 })
((1, 1), Ratio { numer: 1, denom: 2 })
((0, 4), Ratio { numer: 1, denom: 24 })
((0, 3), Ratio { numer: 1, denom: 6 })
((2, 1), Ratio { numer: 1, denom: 8 })
((1, 0), Ratio { numer: 1, denom: 2 })
((0, 1), Ratio { numer: 1, denom: 1 })
((1, 3), Ratio { numer: 1, denom: 12 })
p00 = 48/211 = 0.22748815165876776
probs after normalization:
((1, 2), Ratio { numer: 12, denom: 211 })
((2, 0), Ratio { numer: 6, denom: 211 })
((0, 2), Ratio { numer: 24, denom: 211 })
((3, 0), Ratio { numer: 1, denom: 211 })
((2, 2), Ratio { numer: 3, denom: 211 })
((3, 1), Ratio { numer: 1, denom: 211 })
((1, 1), Ratio { numer: 24, denom: 211 })
((0, 4), Ratio { numer: 2, denom: 211 })
((0, 3), Ratio { numer: 8, denom: 211 })
((2, 1), Ratio { numer: 6, denom: 211 })
((0, 0), Ratio { numer: 48, denom: 211 })
((1, 0), Ratio { numer: 24, denom: 211 })
((0, 1), Ratio { numer: 48, denom: 211 })
((1, 3), Ratio { numer: 4, denom: 211 })
```

Out[13]: ()

Вероятность блокировки запроса на предоставление каждого из типов услуг -- это сумма вероятностей того, что система окажется в одном из блокирующих состояний.

In [14]:
```
println!("B_1 = sum(p(n1, n2)), (n1, n2) \\in B1 = {}", b1.iter().map(|loc|
println!("B_2 = sum(p(n1, n2)), (n1, n2) \\in B2 = {}", b2.iter().map(|loc|
```

```
B_1 = sum(p(n1, n2)), (n1, n2) \in B1 = 11/211
B_2 = sum(p(n1, n2)), (n1, n2) \in B2 = 10/211
```

Среднее число обслуживаемых запросов каждого типа -- это сумма вероятностей оказаться в каждом из состояний, умноженная на количество заявок данного типа, которое обрабатывается в этом состоянии. Для заявок первого типа это первая координата, а для второго типа -- вторая координата.

In [15]:
```
fn rrr(n: i64) -> R {
    R::from_i64(n).unwrap()
}
fn avg_serviced_requests(probs: &HashMap<(i32, i32), R>) -> (R, R) {
    let N1 = probs.iter().map(|((n1, n2), p)| rrr(*n1 as i64) * p).sum::<R>(
    let N2 = probs.iter().map(|((n1, n2), p)| rrr(*n2 as i64) * p).sum::<R>(
    (N1, N2)
```

```
}

let (N1, N2) = avg_serviced_requests(&probs);
println!("N1 = {} = {}", N1, N1.to_f64().unwrap());
println!("N2 = {} = {}", N2, N2.to_f64().unwrap());
```

```
N1 = 100/211 = 0.47393364928909953
N2 = 201/211 = 0.95260663507109
```

Общее среднее число обрабатываемых заявок равно сумме двух разных средних значений.

In [16]:
```
let N = N1 + N2;
println!("N = {} = {}", N, N.to_f64().unwrap());
```

```
N = 301/211 = 1.4265402843601895
```

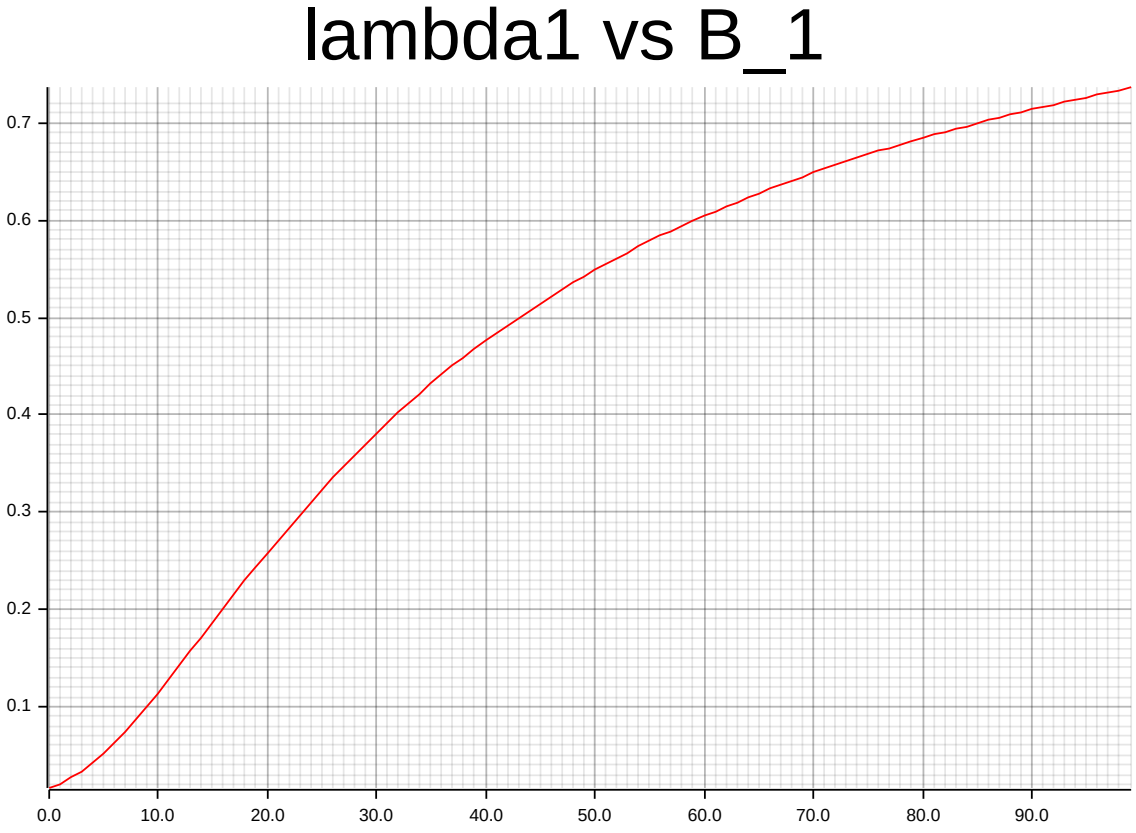
4. графики

График вероятности блокировки заявки можно определить, варьируя значение λ и измеряя вероятность блокировки каждой из заявок. Здесь потребуются четыре графика: два, когда варьируется λ_1 и два для λ_2; в обоих случаях надо рассмотреть вероятность блокировки первого и второго типа заявки.

In [18]:
```
let mut blocking_probs1 = vec![];
let mut blocking_probs2 = vec![];
for lambda1 in 0..100 {
    let rho_1 = lambda1 as f64 / &mu1;
    let probs = make_probs(false, &rho_1, &rho_2, &states);
    let blocking_prob1 = probs.iter().filter(|(v,_)| b1.contains(v)).map(|(_
    let blocking_prob2 = probs.iter().filter(|(v,_)| b2.contains(v)).map(|(_
    blocking_probs1.push((lambda1 as f32, blocking_prob1.to_f32().unwrap()))
    blocking_probs2.push((lambda1 as f32, blocking_prob2.to_f32().unwrap()))
}
println!("{blocking_probs1:?}");
draw_chart(&blocking_probs1, "lambda1 vs B_1")
```

```
[(0.0, 0.015384615), (1.0, 0.020471914), (2.0, 0.026611352), (3.0, 0.0339317
7), (4.0, 0.042455584), (5.0, 0.0521327), (6.0, 0.062866606), (7.0, 0.0745339
7), (8.0, 0.08699891), (9.0, 0.10012309), (10.0, 0.11377245), (11.0, 0.127821
53), (12.0, 0.14215587), (13.0, 0.15667325), (14.0, 0.17128387), (15.0, 0.185
90999), (16.0, 0.20048526), (17.0, 0.21495377), (18.0, 0.22926901), (19.0, 0.
24339284), (20.0, 0.25729442), (21.0, 0.27094936), (22.0, 0.2843387), (23.0,
0.2974483), (24.0, 0.3102679), (25.0, 0.3227907), (26.0, 0.33501273), (27.0,
0.34693238), (28.0, 0.35855), (29.0, 0.36986756), (30.0, 0.38088828), (31.0,
0.3916165), (32.0, 0.40205732), (33.0, 0.41221657), (34.0, 0.42210054), (35.
0, 0.43171585), (36.0, 0.44106945), (37.0, 0.45016837), (38.0, 0.45901984),
(39.0, 0.46763104), (40.0, 0.47600913), (41.0, 0.4841613), (42.0, 0.4920945
5), (43.0, 0.4998158), (44.0, 0.50733185), (45.0, 0.51464933), (46.0, 0.52177
465), (47.0, 0.5287141), (48.0, 0.5354739), (49.0, 0.54205984), (50.0, 0.5484
7777), (51.0, 0.5547331), (52.0, 0.5608313), (53.0, 0.5667775), (54.0, 0.5725
7676), (55.0, 0.57823384), (56.0, 0.58375347), (57.0, 0.58914), (58.0, 0.5943
979), (59.0, 0.59953123), (60.0, 0.60454404), (61.0, 0.6094402), (62.0, 0.614
2234), (63.0, 0.6188973), (64.0, 0.62346524), (65.0, 0.6279306), (66.0, 0.632
2965), (67.0, 0.63656604), (68.0, 0.6407423), (69.0, 0.64482796), (70.0, 0.64
88259), (71.0, 0.6527387), (72.0, 0.6565689), (73.0, 0.660319), (74.0, 0.6639
913), (75.0, 0.66758806), (76.0, 0.6711116), (77.0, 0.6745639), (78.0, 0.6779
471), (79.0, 0.68126315), (80.0, 0.68451387), (81.0, 0.68770117), (82.0, 0.69
082683), (83.0, 0.6938925), (84.0, 0.69689983), (85.0, 0.69985044), (86.0, 0.
70274585), (87.0, 0.7055875), (88.0, 0.70837694), (89.0, 0.7111154), (90.0,
0.71380436), (91.0, 0.71644497), (92.0, 0.7190386), (93.0, 0.7215864), (94.0,
0.7240895), (95.0, 0.72654915), (96.0, 0.72896636), (97.0, 0.7313422), (98.0,
0.7336776), (99.0, 0.7359738)]
```

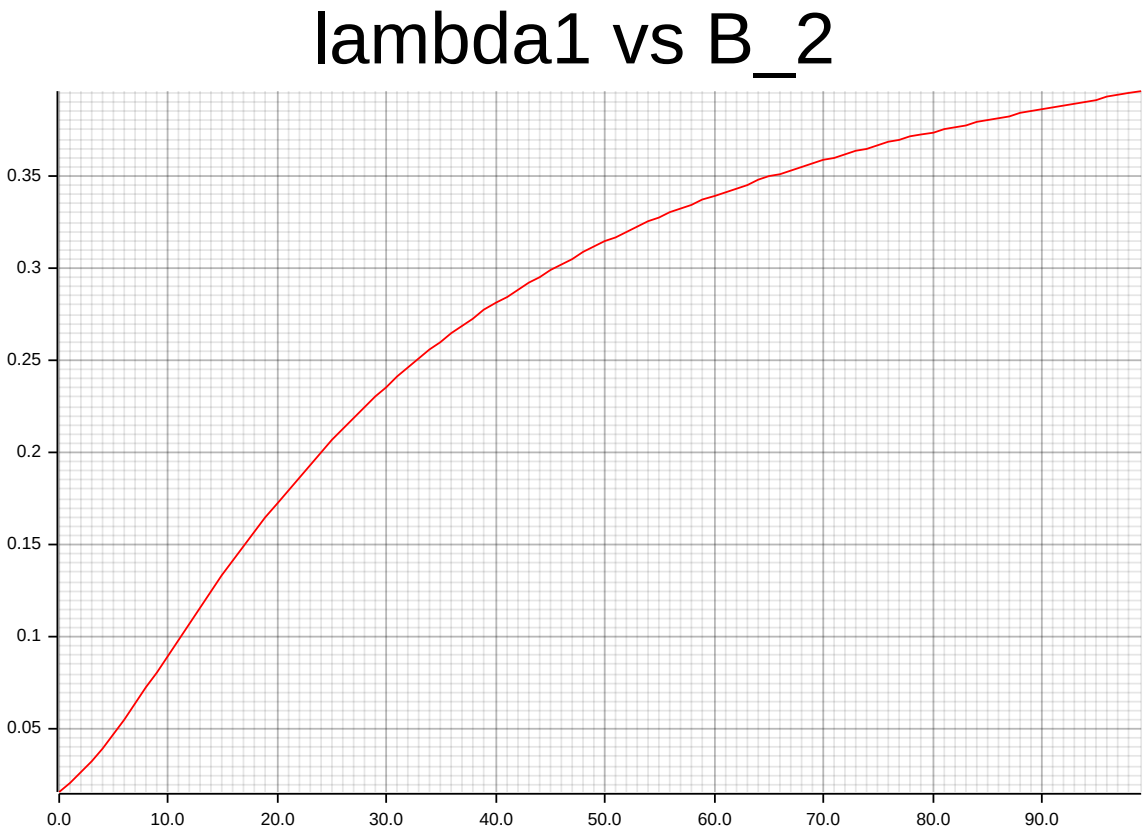
Out[18]:



In [19]:
```
println!("{blocking_probs2:?}");
draw_chart(&blocking_probs2, "lambda1 vs B_2")
```

```
[(0.0, 0.015384615), (1.0, 0.020416131), (2.0, 0.026206618), (3.0, 0.0326920
4), (4.0, 0.03978647), (5.0, 0.047393367), (6.0, 0.055413704), (7.0, 0.063751
53), (8.0, 0.07231749), (9.0, 0.08103082), (10.0, 0.08982036), (11.0, 0.09862
461), (12.0, 0.10739146), (13.0, 0.116077535), (14.0, 0.124647334), (15.0, 0.
1330724), (16.0, 0.1413304), (17.0, 0.14940427), (18.0, 0.15728146), (19.0,
0.16495317), (20.0, 0.1724138), (21.0, 0.17966032), (22.0, 0.18669185), (23.
0, 0.19350925), (24.0, 0.20011474), (25.0, 0.20651163), (26.0, 0.21270406),
(27.0, 0.21869686), (28.0, 0.22449528), (29.0, 0.23010492), (30.0, 0.2355316
3), (31.0, 0.24078134), (32.0, 0.24586007), (33.0, 0.25077382), (34.0, 0.2555
2854), (35.0, 0.26013008), (36.0, 0.26458415), (37.0, 0.26889643), (38.0, 0.2
730723), (39.0, 0.27711707), (40.0, 0.2810358), (41.0, 0.28483343), (42.0, 0.
2885147), (43.0, 0.29208416), (44.0, 0.2955462), (45.0, 0.29890501), (46.0,
0.3021646), (47.0, 0.30532888), (48.0, 0.30840147), (49.0, 0.31138596), (50.
0, 0.31428573), (51.0, 0.31710398), (52.0, 0.31984383), (53.0, 0.32250825),
(54.0, 0.32510003), (55.0, 0.3276219), (56.0, 0.3300765), (57.0, 0.33246619),
(58.0, 0.3347934), (59.0, 0.33706036), (60.0, 0.33926928), (61.0, 0.3414221
7), (62.0, 0.34352106), (63.0, 0.34556782), (64.0, 0.34756425), (65.0, 0.3495
121), (66.0, 0.35141304), (67.0, 0.35326862), (68.0, 0.35508043), (69.0, 0.35
684988), (70.0, 0.35857838), (71.0, 0.36026728), (72.0, 0.36191785), (73.0,
0.36353135), (74.0, 0.36510897), (75.0, 0.3666518), (76.0, 0.368161), (77.0,
0.36963755), (78.0, 0.37108248), (79.0, 0.37249678), (80.0, 0.3738814), (81.
0, 0.37523717), (82.0, 0.37656498), (83.0, 0.37786567), (84.0, 0.37914005),
(85.0, 0.38038886), (86.0, 0.3816128), (87.0, 0.38281268), (88.0, 0.3839891),
(89.0, 0.38514277), (90.0, 0.38627428), (91.0, 0.3873843), (92.0, 0.3884734),
(93.0, 0.3895421), (94.0, 0.39059103), (95.0, 0.3916207), (96.0, 0.3926316),
(97.0, 0.39362422), (98.0, 0.39459905), (99.0, 0.39555657)]
```

Out[19]:



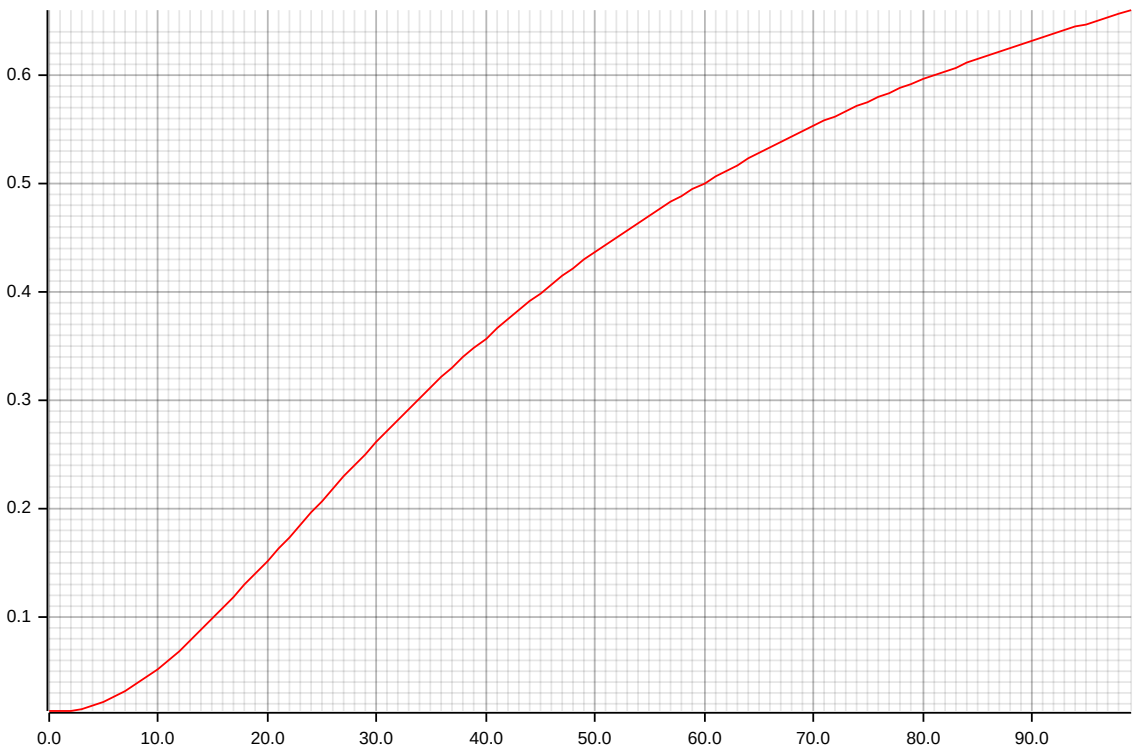
In [20]:

```
let mut blocking_probs1 = vec![];
let mut blocking_probs2 = vec![];
for lambda2 in 0..100 {
```

```
    let rho_2 = lambda2 as f64 / &mu1;
    let probs = make_probs(false, &rho_1, &rho_2, &states);
    let blocking_prob1 = probs.iter().filter(|(v,_)| b1.contains(v)).map(|(_
    let blocking_prob2 = probs.iter().filter(|(v,_)| b2.contains(v)).map(|(_
    blocking_probs1.push((lambda2 as f32, blocking_prob1.to_f32().unwrap()))
    blocking_probs2.push((lambda2 as f32, blocking_prob2.to_f32().unwrap()))
}
println!("{blocking_probs1:?}");
draw_chart(&blocking_probs1, "lambda2 vs B_1")
```

```
[(0.0, 0.012658228), (1.0, 0.012991655), (2.0, 0.014049463), (3.0, 0.01590048
7), (4.0, 0.018589282), (5.0, 0.02213667), (6.0, 0.026541755), (7.0, 0.031784
83), (8.0, 0.037830684), (9.0, 0.044632025), (10.0, 0.0521327), (11.0, 0.0602
7065), (12.0, 0.068980455), (13.0, 0.07819549), (14.0, 0.08784965), (15.0, 0.
09787867), (16.0, 0.10822119), (17.0, 0.11881937), (18.0, 0.12961943), (19.0,
0.14057185), (20.0, 0.15163147), (21.0, 0.16275752), (22.0, 0.17391339), (23.
0, 0.1850665), (24.0, 0.19618814), (25.0, 0.20725307), (26.0, 0.21823932), (2
7.0, 0.22912799), (28.0, 0.23990281), (29.0, 0.25055003), (30.0, 0.26105812),
(31.0, 0.27141747), (32.0, 0.28162032), (33.0, 0.29166043), (34.0, 0.3015329
2), (35.0, 0.31123418), (36.0, 0.3207616), (37.0, 0.33011356), (38.0, 0.33928
925), (39.0, 0.34828848), (40.0, 0.35711178), (41.0, 0.36576012), (42.0, 0.37
42349), (43.0, 0.38253796), (44.0, 0.3906714), (45.0, 0.39863762), (46.0, 0.4
0643916), (47.0, 0.41407883), (48.0, 0.42155954), (49.0, 0.4288843), (50.0,
0.4360562), (51.0, 0.4430784), (52.0, 0.4499541), (53.0, 0.45668653), (54.0,
0.46327886), (55.0, 0.46973437), (56.0, 0.47605622), (57.0, 0.4822476), (58.
0, 0.48831165), (59.0, 0.49425143), (60.0, 0.50007004), (61.0, 0.5057704), (6
2.0, 0.5113555), (63.0, 0.5168283), (64.0, 0.5221915), (65.0, 0.52744794), (6
6.0, 0.5326003), (67.0, 0.5376512), (68.0, 0.5426032), (69.0, 0.5474588), (7
0.0, 0.5522206), (71.0, 0.5568908), (72.0, 0.56147176), (73.0, 0.5659658), (7
4.0, 0.570375), (75.0, 0.57470167), (76.0, 0.5789477), (77.0, 0.5831153), (7
8.0, 0.5872063), (79.0, 0.59122264), (80.0, 0.59516615), (81.0, 0.5990387),
(82.0, 0.602842), (83.0, 0.6065777), (84.0, 0.61024755), (85.0, 0.6138531),
(86.0, 0.61739594), (87.0, 0.6208775), (88.0, 0.62429935), (89.0, 0.6276628
4), (90.0, 0.63096946), (91.0, 0.6342204), (92.0, 0.63741714), (93.0, 0.64056
087), (94.0, 0.6436528), (95.0, 0.6466941), (96.0, 0.6496861), (97.0, 0.65262
973), (98.0, 0.6555262), (99.0, 0.65837663)]
```

Out[20]:



In [21]:
```
println!("{blocking_probs2:?}");
draw_chart(&blocking_probs2, "lambda2 vs B_2")
```

```
[(0.0, 0.0), (1.0, 0.0015371892), (2.0, 0.003682386), (3.0, 0.006515239), (4.
0, 0.010090158), (5.0, 0.014436958), (6.0, 0.01956296), (7.0, 0.025455963),
(8.0, 0.032087624), (9.0, 0.03941691), (10.0, 0.047393367), (11.0, 0.05596012
2), (12.0, 0.06505647), (13.0, 0.07462005), (14.0, 0.08458862), (15.0, 0.0949
01375), (16.0, 0.105500095), (17.0, 0.11632977), (18.0, 0.12733912), (19.0,
0.13848095), (20.0, 0.14971209), (21.0, 0.16099359), (22.0, 0.1722905), (23.
0, 0.1835717), (24.0, 0.19480975), (25.0, 0.2059806), (26.0, 0.21706334), (2
7.0, 0.22803995), (28.0, 0.23889506), (29.0, 0.24961564), (30.0, 0.2601908),
(31.0, 0.2706116), (32.0, 0.28087077), (33.0, 0.29096252), (34.0, 0.3008824
6), (35.0, 0.3106273), (36.0, 0.3201949), (37.0, 0.32958385), (38.0, 0.338793
67), (39.0, 0.34782442), (40.0, 0.35667682), (41.0, 0.36535206), (42.0, 0.373
8518), (43.0, 0.38217795), (44.0, 0.39033282), (45.0, 0.39831892), (46.0, 0.4
06139), (47.0, 0.41379586), (48.0, 0.42129257), (49.0, 0.42863223), (50.0, 0.
43581805), (51.0, 0.44285324), (52.0, 0.44974107), (53.0, 0.45648482), (54.0,
0.46308777), (55.0, 0.4695532), (56.0, 0.47588438), (57.0, 0.48208445), (58.
0, 0.48815668), (59.0, 0.49410418), (60.0, 0.49993), (61.0, 0.50563717), (62.
0, 0.5112287), (63.0, 0.5167075), (64.0, 0.52207637), (65.0, 0.52733815), (6
6.0, 0.5324955), (67.0, 0.53755116), (68.0, 0.54250765), (69.0, 0.5473676),
(70.0, 0.5521333), (71.0, 0.55680734), (72.0, 0.5613919), (73.0, 0.5658893),
(74.0, 0.57030183), (75.0, 0.5746315), (76.0, 0.5788805), (77.0, 0.58305085),
(78.0, 0.58714443), (79.0, 0.5911633), (80.0, 0.59510916), (81.0, 0.5989839
4), (82.0, 0.6027894), (83.0, 0.60652715), (84.0, 0.6101989), (85.0, 0.613806
3), (86.0, 0.6173509), (87.0, 0.6208342), (88.0, 0.6242576), (89.0, 0.6276226
6), (90.0, 0.6309307), (91.0, 0.6341831), (92.0, 0.63738114), (93.0, 0.640526
1), (94.0, 0.64361924), (95.0, 0.6466618), (96.0, 0.64965487), (97.0, 0.65259
96), (98.0, 0.65549713), (99.0, 0.6583485)]
```

Out[21]:

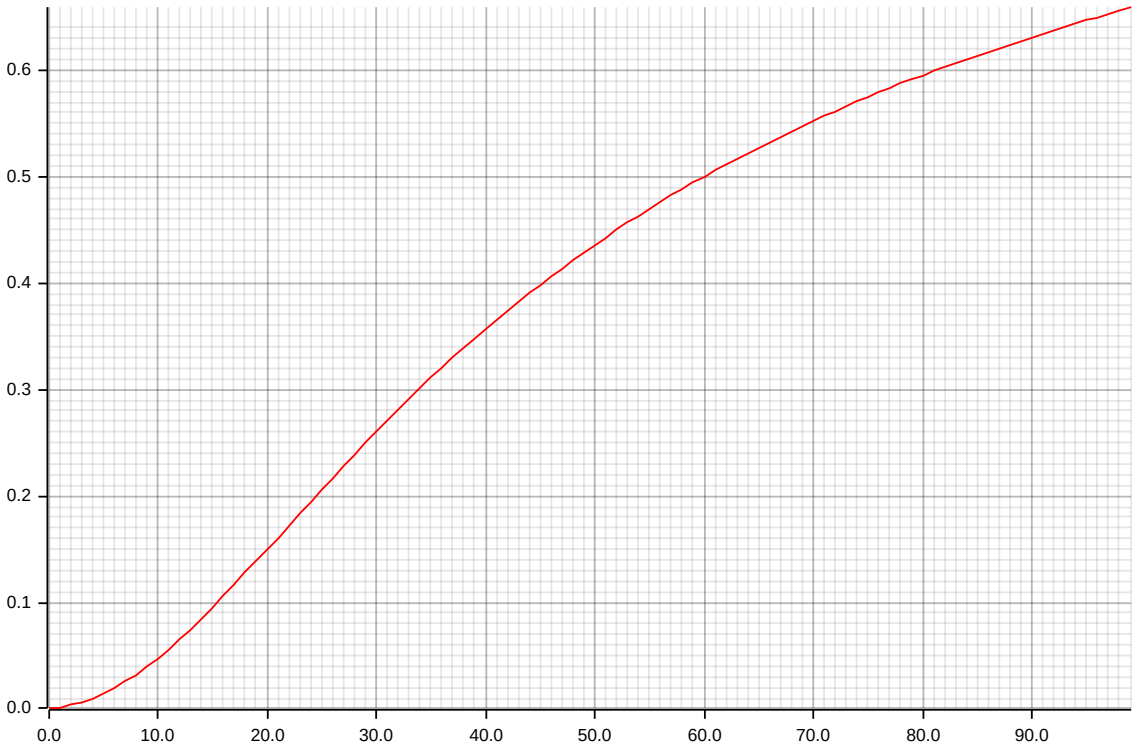


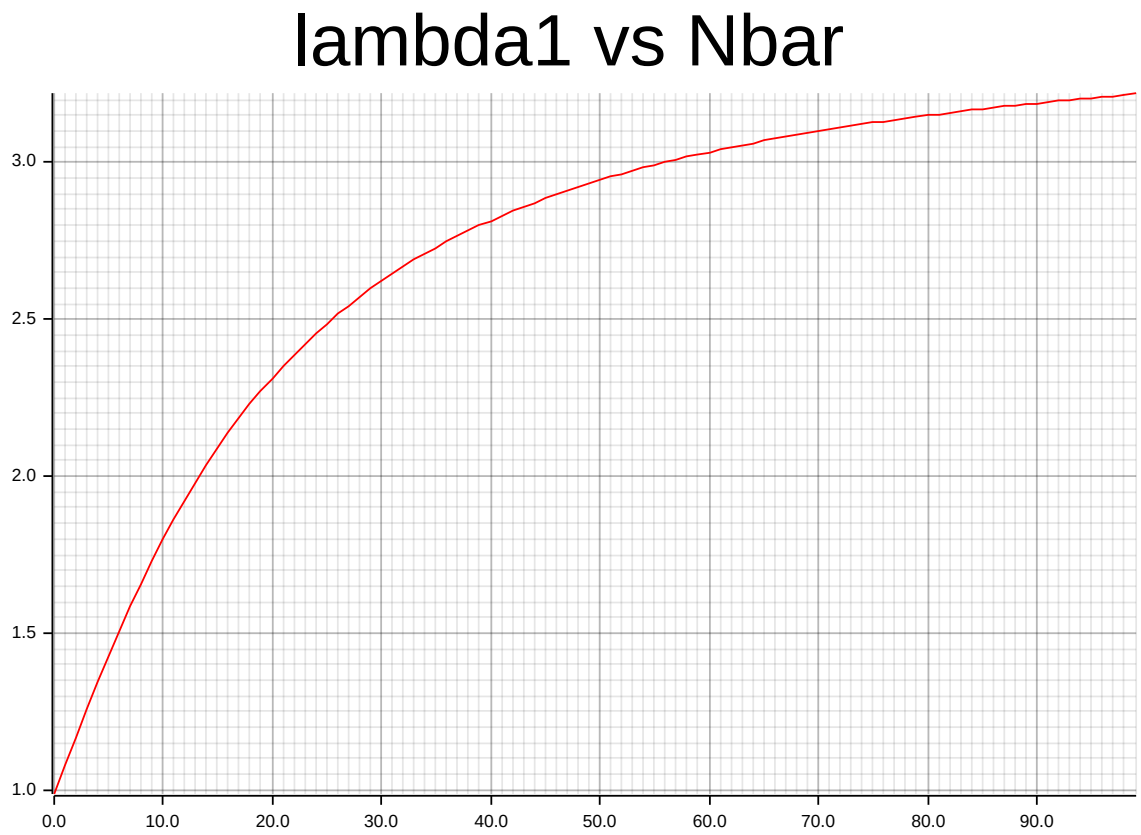
График зависимости среднего числа обслуживаемых заявок можно вычислить похожим образом: варьировать интенсивность поступления заявок и измерять среднее число. Здесь можно обойтись двумя графиками, если суммировать количество обслуживаемых заявок между двумя типами.

In [22]:

```
let mut avg_svc = vec![];
for lambda1 in 0..100 {
    let rho_1 = lambda1 as f64 / &mu1;
    let probs = make_probs(false, &rho_1, &rho_2, &states);
    let reqs = avg_serviced_requests(&probs);
    let avg_service = reqs.0 + reqs.1;
    avg_svc.push((lambda1 as f32, avg_service.to_f32().unwrap()));
}
println!("{avg_svc:?}");
draw_chart(&avg_svc, "lambda1 vs Nbar")
```

```
[(0.0, 0.9846154), (1.0, 1.0775367), (2.0, 1.1684711), (3.0, 1.2571285), (4.
0, 1.3432313), (5.0, 1.4265403), (6.0, 1.5068663), (7.0, 1.5840747), (8.0, 1.
6580834), (9.0, 1.7288584), (10.0, 1.7964072), (11.0, 1.8607717), (12.0, 1.92
20215), (13.0, 1.9802473), (14.0, 2.0355554), (15.0, 2.0880625), (16.0, 2.137
8932), (17.0, 2.1851742), (18.0, 2.2300344), (19.0, 2.2726004), (20.0, 2.3129
973), (21.0, 2.351346), (22.0, 2.387763), (23.0, 2.4223597), (24.0, 2.455242
4), (25.0, 2.4865117), (26.0, 2.5162628), (27.0, 2.5445857), (28.0, 2.571564
7), (29.0, 2.597279), (30.0, 2.6218035), (31.0, 2.6452074), (32.0, 2.667556
5), (33.0, 2.6889114), (34.0, 2.7093296), (35.0, 2.7288644), (36.0, 2.747565
7), (37.0, 2.7654805), (38.0, 2.7826524), (39.0, 2.7991219), (40.0, 2.814927
6), (41.0, 2.8301053), (42.0, 2.8446882), (43.0, 2.858708), (44.0, 2.872193
6), (45.0, 2.885173), (46.0, 2.897672), (47.0, 2.9097147), (48.0, 2.9213238),
(49.0, 2.9325206), (50.0, 2.9433255), (51.0, 2.953757), (52.0, 2.9638333), (5
3.0, 2.9735708), (54.0, 2.9829855), (55.0, 2.992092), (56.0, 3.0009043), (57.
0, 3.0094357), (58.0, 3.0176988), (59.0, 3.0257053), (60.0, 3.0334663), (61.
0, 3.0409925), (62.0, 3.0482936), (63.0, 3.0553792), (64.0, 3.0622582), (65.
0, 3.0689392), (66.0, 3.0754302), (67.0, 3.0817387), (68.0, 3.087872), (69.0,
3.093837), (70.0, 3.0996404), (71.0, 3.105288), (72.0, 3.1107862), (73.0, 3.1
161404), (74.0, 3.1213558), (75.0, 3.126438), (76.0, 3.131391), (77.0, 3.1362
205), (78.0, 3.1409302), (79.0, 3.1455245), (80.0, 3.1500075), (81.0, 3.15438
32), (82.0, 3.158655), (83.0, 3.1628265), (84.0, 3.1669014), (85.0, 3.170882
5), (86.0, 3.174773), (87.0, 3.1785758), (88.0, 3.182294), (89.0, 3.18593),
(90.0, 3.1894867), (91.0, 3.1929665), (92.0, 3.1963716), (93.0, 3.1997044),
(94.0, 3.2029674), (95.0, 3.2061625), (96.0, 3.2092915), (97.0, 3.2123568),
(98.0, 3.21536), (99.0, 3.218303)]
```

Out[22]:



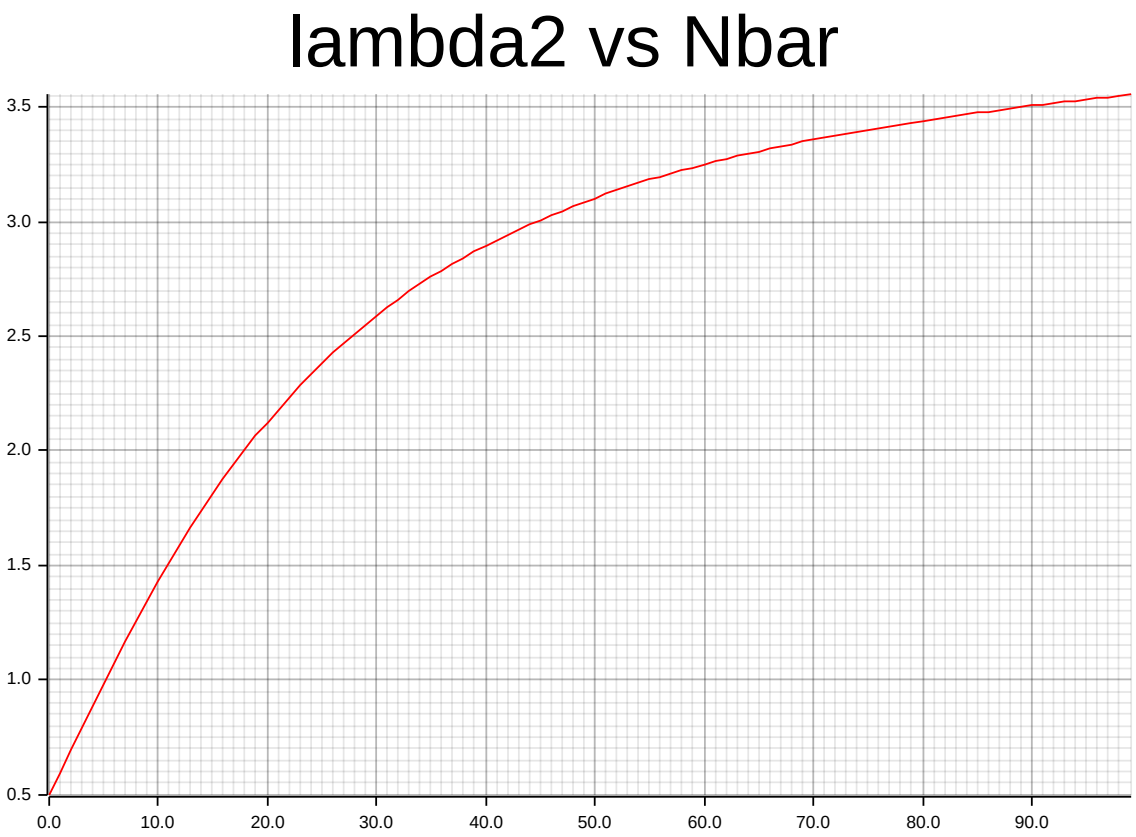
In [25]:

```
let mut avg_svc = vec![];
for lambda2 in 0..100 {
    let rho_2 = lambda2 as f64 / &mu1;
    let probs = make_probs(false, &rho_1, &rho_2, &states);
```

```
    let reqs = avg_serviced_requests(&probs);
    let avg_service = reqs.0 + reqs.1;
    avg_svc.push((lambda2 as f32, avg_service.to_f32().unwrap()));
}
println!("{avg_svc:?}");
draw_chart(&avg_svc, "lambda2 vs Nbar")
```

```
[(0.0, 0.49367088), (1.0, 0.59335047), (2.0, 0.6922388), (3.0, 0.7900952),
(4.0, 0.8866693), (5.0, 0.9817132), (6.0, 1.0749913), (7.0, 1.1662884), (8.0,
1.2554146), (9.0, 1.3422087), (10.0, 1.4265403), (11.0, 1.5083085), (12.0, 1.
587442), (13.0, 1.6638962), (14.0, 1.7376511), (15.0, 1.8087085), (16.0, 1.87
70893), (17.0, 1.9428297), (18.0, 2.0059798), (19.0, 2.0666003), (20.0, 2.124
7602), (21.0, 2.1805346), (22.0, 2.2340043), (23.0, 2.2852519), (24.0, 2.3343
625), (25.0, 2.381422), (26.0, 2.4265156), (27.0, 2.4697282), (28.0, 2.511142
5), (29.0, 2.5508397), (30.0, 2.5888984), (31.0, 2.6253953), (32.0, 2.660403
5), (33.0, 2.6939936), (34.0, 2.7262332), (35.0, 2.7571874), (36.0, 2.786917
4), (37.0, 2.8154829), (38.0, 2.8429394), (39.0, 2.8693407), (40.0, 2.894736
8), (41.0, 2.9191763), (42.0, 2.942705), (43.0, 2.965366), (44.0, 2.9871998),
(45.0, 3.008246), (46.0, 3.028541), (47.0, 3.04812), (48.0, 3.067016), (49.0,
3.08526), (50.0, 3.1028817), (51.0, 3.1199093), (52.0, 3.1363695), (53.0, 3.1
522872), (54.0, 3.1676867), (55.0, 3.1825902), (56.0, 3.1970193), (57.0, 3.21
09947), (58.0, 3.2245355), (59.0, 3.2376597), (60.0, 3.250385), (61.0, 3.2627
28), (62.0, 3.2747045), (63.0, 3.2863288), (64.0, 3.2976155), (65.0, 3.308578
3), (66.0, 3.3192296), (67.0, 3.3295817), (68.0, 3.3396463), (69.0, 3.349434
4), (70.0, 3.3589566), (71.0, 3.3682227), (72.0, 3.3772426), (73.0, 3.38602
5), (74.0, 3.394579), (75.0, 3.4029129), (76.0, 3.4110343), (77.0, 3.418951),
(78.0, 3.4266703), (79.0, 3.4341989), (80.0, 3.4415436), (81.0, 3.4487107),
(82.0, 3.4557061), (83.0, 3.4625359), (84.0, 3.4692051), (85.0, 3.4757197),
(86.0, 3.4820843), (87.0, 3.488304), (88.0, 3.4943833), (89.0, 3.5003269), (9
0.0, 3.5061388), (91.0, 3.5118237), (92.0, 3.517385), (93.0, 3.5228267), (94.
0, 3.5281525), (95.0, 3.533366), (96.0, 3.5384703), (97.0, 3.543469), (98.0,
3.548365), (99.0, 3.5531616)]
```

Out[25]:


lambda2 vs Nbar
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0

In [ ]:

Отчет по лабораторной работе 2

Неполнодоступная двухсервисная модель Эрланга с одинаковыми интенсивностями обслуживания и зарезервированной емкостью

Генералов Даниил, 1032212280

0. теоретическая информация

Исследуется сота сети связи емкостью C. Пусть пользователям сети предоставляются услуги двух типов. Запросы в виде двух пуассоновский потоков (ПП) с интенсивностями λ_1, λ_2 поступают в соту. Среднее время обслуживания запросов на предоставление услуг каждого типа μ_1^{-1}, μ_2^{-1} соответственно. Исследуются основные характеристики модели для случая $\mu_1 = \mu_2 = \mu$.

Часть пропускной способности соты зарезервирована для обслуживания запросов на предоставление услуги 1-го или 2-го типа. Оставшаяся часть пропускной способности является полнодоступной для запросов на предоставление услуг обоих типов. Предположим, что сначала заполняется полнодоступная емкость.

В классификации Башарина-Кендалла: $M(\lambda_1)M(\lambda_2)|M(\mu_1)M(\mu_2)|C, g|0$.

- C -- пиковая пропускная способность соты;
- g -- полнодоступная часть пропускной способности соты;
- $C - g$ -- пропускная способность, зарезервированная для обслуживание запросов определенного типа;
- λ_1, λ_2 -- интенсивность поступления запросов на предоставление услуги 1, 2-го типа;
- μ^{-1} -- среднее время обслуживания запроса на предоставление услуги 1, 2-го типа;
- ρ_1, ρ_2 -- интенсивность предложенной нагрузки, создаваемой запросами на предоставление услуги 1, 2-го типа;
- $X(t)$ -- число запросов, обслуживаемых в системе в момент времени t, $t \geq 0$ (случайный процесс (СП), описывающий функционирование системы в момент времени t, $t \geq 0$);
- X -- пространство состояний системы;
- n -- число обслуживаемых в системе запросов;

- B_1, B_2 -- множество блокировок запросов на предоставление услуги 1, 2-го типа;
- S_1, S_2 -- множество приема запросов на предоставление услуги 1, 2-го типа.

Пусть часть пропускной способности соты $\mathrm{C} - g$ зарезервирована для обслуживания запросов на предоставление услуги 1-го типа.

Пространство состояний системы: $X = \{0, \ldots, C\}$; $|X| = C + 1$

Множество блокировок запросов на предоставление услуги i-типа, $i = 1, 2$: $B_1\{C\}$; $B_2 = \{g, g+1, \ldots, C\}$

Множество приема запросов на предоставление услуги i-типа, $i = 1, 2$: $S_i = \overline{B_i} = X \setminus B_i$; $S_1 = \{0, \ldots, C-1\}$; $S_2 = \{0, \ldots, C_2 - 1\}$

СУГБ:

- $(\lambda_1 + \lambda_2)p_0 = \mu p_1$
- $(\lambda_1 + \lambda_2 + n\mu)p_n = (\lambda_1 + \lambda_2)p_{n-1} + (n+1)\mu p_{n+1}$, $n = \overline{1, g-1}$
- $(\lambda_1 + g\mu)p_g = (\lambda_1 + \lambda_2)p_{g-1} + (g+1)\mu p_{g+1}$
- $(\lambda_1 + n\mu)p_n = \lambda_1 p_{n-1} + (n+1)\mu p_{n+1}$, $n = \overline{g+1, C-1}$
- $C\mu p_C = \lambda_1 p_{C-1}$

СУЛБ:

- $(\lambda_1 + \lambda_2)p_1 = n\mu p_n$, $n = \overline{1, g}$
- $\lambda_1 p_{n-1} = n\mu p_n$, $n = \overline{g+1, C}$

Обозначим $\rho_1 = \frac{\lambda_1}{\mu}$, $\rho_2 = \frac{\lambda_2}{\mu}$.

Стационарное распределение вероятностей состояний системы:

$p_n =$

- $p_0 * \frac{(\rho_1+\rho_2)^n}{n!}$, если $n = \overline{1, g}$;
- $p_0 * \frac{(\rho_1+\rho_2)^g * (\rho_1)^{n-g}}{n!}$, если $n = \overline{g+1, C}$.

При этом $p_0 = (\sum_{n=1}^{g} \frac{(\rho_1+\rho_2)^n}{n!} + \sum_{n=g+1}^{C} \frac{(\rho_1+\rho_2)^g * (\rho_1)^{n-g}}{n!})^{-1}$

Основные вероятностные характеристики модели:

- Вероятность блокировки по времени запроса на предоставление услуги 1-го типа: $E_1 = \sum_{n \in B_1} p_n = p_C$
- Вероятность блокировки по времени запроса на предоставление услуги 1-го типа: $E_2 = \sum_{n \in B_2} p_n = p_g + p_{g+1} + \ldots + p_C = \sum_{n=g}^{C} p_n$

- Среднее число обслуживаемых в системе запросов: $\overline{N} = \sum_{n \in X} np_n$

1. подключение библиотек, определение функций

Для расчета больших факториалов нам потребуется длинная арифметика, а для рисования графиков -- библиотека для визуализации данных.

In [2]:
```
:dep num = { version = "^0.4.3" }
:dep plotters = { version = "^0.3.6", default-features = false, features = [

extern crate num;
use num::BigRational as R;
use num::BigInt as I;
use num::BigUint as U;
use num::Integer;
use num::traits::ConstZero;
use num::FromPrimitive;
use num::ToPrimitive;

extern crate plotters;
use plotters::prelude::*;
```

Для удобства конвертации стандартных чисел в числа длинной арифметики используются helper-функции.

In [3]:
```
fn u(i: usize) -> U {
    U::from_usize(i).unwrap()
}

fn rr(i: f64) -> R {
    R::from_float(i).unwrap()
}
```

Для вычисления факториала нет стандартной функции, и очевидные подходы не работают с длинной арифметикой, поэтому эта функция считает это за нас.

Мы также проверяем, работает ли одно из правил факториалов: что $N! = (N-i)! + \prod_{x=N-i+1}^{N} x$.

In [4]:
```
use num::{BigInt, BigUint};
use num::ToPrimitive;
fn factorial(n: u8) -> BigUint {
    (1..=n).map(BigUint::from).product()
}

for i in 3..8 {
    assert_eq!(factorial(10), factorial(i) * (i+1 ..= 10).map(|x| x as usize
}
```

Out[4]: ()

Эта функция отображает график функции, принимая на вход список X-Y пар.

In [5]:
```
fn draw_chart(data: &Vec<(f32, f32)>, name: impl ToString) -> plotters::evcx
    let minx = data.iter().min_by(|a, b| a.0.partial_cmp(&b.0).unwrap_or(std
    let maxx = data.iter().max_by(|a, b| a.0.partial_cmp(&b.0).unwrap_or(std
    let miny = data.iter().min_by(|a, b| a.1.partial_cmp(&b.1).unwrap_or(std
    let maxy = data.iter().max_by(|a, b| a.1.partial_cmp(&b.1).unwrap_or(std
    let figure = evcxr_figure((640, 480), |root| {
        root.fill(&WHITE)?;
        let mut chart = ChartBuilder::on(&root)
            .caption(name.to_string(), ("Arial", 50).into_font())
            .margin(5)
            .x_label_area_size(30)
            .y_label_area_size(30)
            .build_cartesian_2d(minx..maxx, miny..maxy)?;

        chart.configure_mesh().draw()?;

        chart.draw_series(LineSeries::new(
            data.clone(),
            &RED,
        )).unwrap();

        // chart.configure_series_labels()
        //     .background_style(&WHITE.mix(0.8))
        //     .border_style(&BLACK)
        //     .draw()?;
        Ok(())
    });
    return figure;
}
```

Эта функция считает стационарное распределение вероятностей для рассматриваемой модели. Она принимает ρ_1 и ρ_2, которые соответствуют интенсивностям поступления заявок на два типа услуг, а также C -- максимальную пропускную способность, а именно количество заявок, которые могут одновременно выполняться, и g -- зарезервированная емкость для заявок первого типа.

Подсчет разделен на два случая: `prob_low` и `prob_high` -- первое относится к случаю, когда $i < g$, а второе к остальным случаям. Из-за этого `prob_low` используется в определении `prob_high`.

Функция возвращает список: i-й элемент списка равен вероятности, что система будет найдена в состоянии i (то есть это p_i).

In [6]:
```
/// Считает стационарное распределение вероятностей для пространства
fn stationary_prob_distribution(rho1: f64, rho2: f64, max_c: u8, g: u8) -> V
    let prob_low = |x| (rho1 + rho2).powf(x as f64) / (factorial(x).to_f64()
    let prob_high = |x| (
```

```
        prob_low(g) *
        rho1.powi(x as i32 - g as i32) /
        ((g+1) as usize..=x as usize).map(|x| x as f64).product::<f64>()
    );
    let prob = |x| if x<=g {prob_low(x)} else {prob_high(x)};

    let mut v = Vec::with_capacity(max_c as usize);
    let p0 = 1.0/(1..=max_c)
        // .map(|x| (println!("{x}"), x).1)
        .map(prob).sum::<f64>();

    for c in 0..=max_c {
        let res = p0 * prob(c);
        v.push(res);
    }
    v
}
```

2. входные параметры

Здесь задаются параметры, которые определяют модель. Чтобы попробовать запустить вычисления с другими значениями, вы можете поменять эту ячейку и перезапустить ее и все ячейки ниже.

При разработке я использовал следующие значения для теста:

- $\lambda 1 = 80$
- $\lambda 2 = 50$
- $\mu_1 = \mu_2 = 2$
- $C = 60$
- $g = 40$

In [7]:
```
let lambda1 = 80.0;
let lambda2 = 50.0;
let mu = 2.0;
let mu1 = mu;
let mu2 = mu;
let rho1 = lambda1 / mu1;
let rho2 = lambda2 / mu2;
let c = 60;
let g: usize = 40;
```

3. расчет

Распределение вероятностей можно посчитать с помощью нашей функции сверху. По свойству распределения вероятностей, сумма всех элементов должна быть равна 1, и мы здесь проверяем это.

In [9]:
```
let dist = stationary_prob_distribution(rho1, rho2, c as u8, g as u8);
println!("probability distribution: {dist:?}");
println!("Sum should be 1: {}", dist.iter().sum::<f64>());
```

```
probability distribution: [2.5542482810751626e-26, 1.6602613826988556e-24, 5.
395849493771281e-23, 1.1691007236504442e-21, 1.899788675931972e-20, 2.4697252
78711563e-19, 2.6755357186041938e-18, 2.4844260244181796e-17, 2.0185961448397
71e-16, 1.4578749934953903e-15, 9.476187457720037e-15, 5.599565315925476e-14,
3.0330978794596327e-13, 1.5165489397298164e-12, 7.041120077317004e-12, 3.0511
520335040355e-11, 1.2395305136110144e-10, 4.739381375571525e-10, 1.7114432745
119397e-9, 5.854937518067162e-9, 1.9028546933718278e-8, 5.889788336627086e-8,
1.7401647358216393e-7, 4.917856862104632e-7, 1.3319195668200045e-6, 3.4629908
737320113e-6, 8.657477184330028e-6, 2.084207470301674e-5, 4.838338770343171e-
5, 0.00010844552416286419, 0.00023496530235287238, 0.000492669182352797, 0.00
1000734276654119, 0.001971143272197507, 0.0037683621380246446, 0.006998386827
7600555, 0.01263597621678899, 0.022198336597061736, 0.03797083891602666, 0.06
32847315267111, 0.10283768873090554, 0.10032945242039565, 0.0955518594479958
4, 0.08888545064929847, 0.08080495513572587, 0.07182662678731189, 0.062457936
336792946, 0.05315569049939826, 0.04429640874949854, 0.03616033367306004, 0.0
28928266938448032, 0.02268883681446904, 0.017452951395745417, 0.0131720387892
41823, 0.009757065769808759, 0.007096047832588188, 0.005068605594705848, 0.00
35569162068111214, 0.002453045659869739, 0.001663081803301518, 0.001108721202
201012]
Sum should be 1: 1
```

Среднее число запросов, которые находятся в обработке -- это математическое ожидание набора вероятностей: ((вероятность того, что там 1 заявка) * 1 + (вероятность того, что там 2 заявки) * 2 + ...) / (макс. заявок)

Чтобы посчитать это, мы считаем сумму значений вероятности, помноженных на свой индекс (который равен количеству заявок, соответствующих этой ячейке).

In [10]:
```
let avg: f64 = dist.iter().enumerate().map(|(i,v)| (i as f64) * v).sum();
println!("Average requests in flight: {avg}");
```

```
Average requests in flight: 43.72435164097261
```

Вероятность блокировки по времени -- это вероятность того, что система оказывается в состоянии, когда нельзя обработать ни одну заявку, пока нагрузка не спадет.

Для заявок первого типа это -- только случай, когда нет больше ячеек вообще (то есть p_C), а для второго типа это все случаи, когда обрабатывается больше чем g заявок (потому что та пропускная способность зарезервирована для заявок первого типа).

In [11]:
```
let time_blocking_prob1 = dist.last().copied().unwrap();
let time_blocking_prob2 = dist.iter().skip(g).sum::<f64>();
println!("time blocking probability (E1): {time_blocking_prob1}");
println!("time blocking probability (E2): {time_blocking_prob2}");
```

```
time blocking probability (E1): 0.001108721202201012
time blocking probability (E2): 0.8492519804375737
```

Вероятность блокировки по вызовам -- это вероятность, что определенный из двух типов запросов будет заблокирован; поэтому это связано с вероятностью поступления каждого из двух запросов.

In [12]:
```
let req_blocking_prob1 = lambda1 / (lambda1 + lambda2) * time_blocking_prob1
let req_blocking_prob2 = lambda2 / (lambda1 + lambda2) * time_blocking_prob1
println!("request blocking probability: {req_blocking_prob1}, {req_blocking_
```

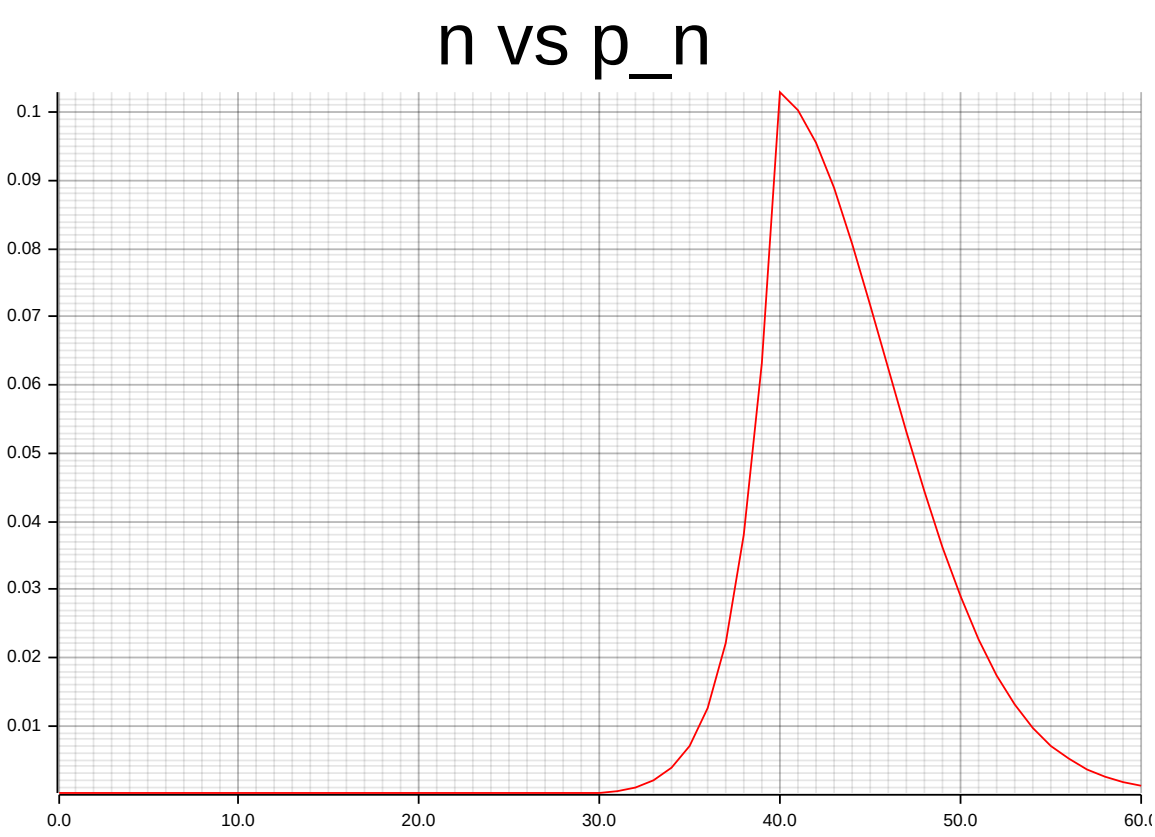
```
request blocking probability: 0.0006822899705852383, 0.0004264312316157739
```

4. графики

График распределения вероятности показывает, какие состояния системы являются самыми частыми. Если пик этого графика находится на правой границе, значит система перегружена; иначе он показывает среднее количество ожидающих своей очереди запросов (таким образом это значение имеет похожий смысл на E).

In [13]:
```
draw_chart(&dist.iter().enumerate().map(|(x,y)| (x as f32,*y as f32)).collec
```

Out[13]:



Для того, чтобыв посчитать график зависимости вероятности блокировки от интенсивности, надо зафиксировать значение одной из переменных λ и

варьировать другую; для каждой такой конфигурации нужно вычислить значение E и затем отобразить на графике.

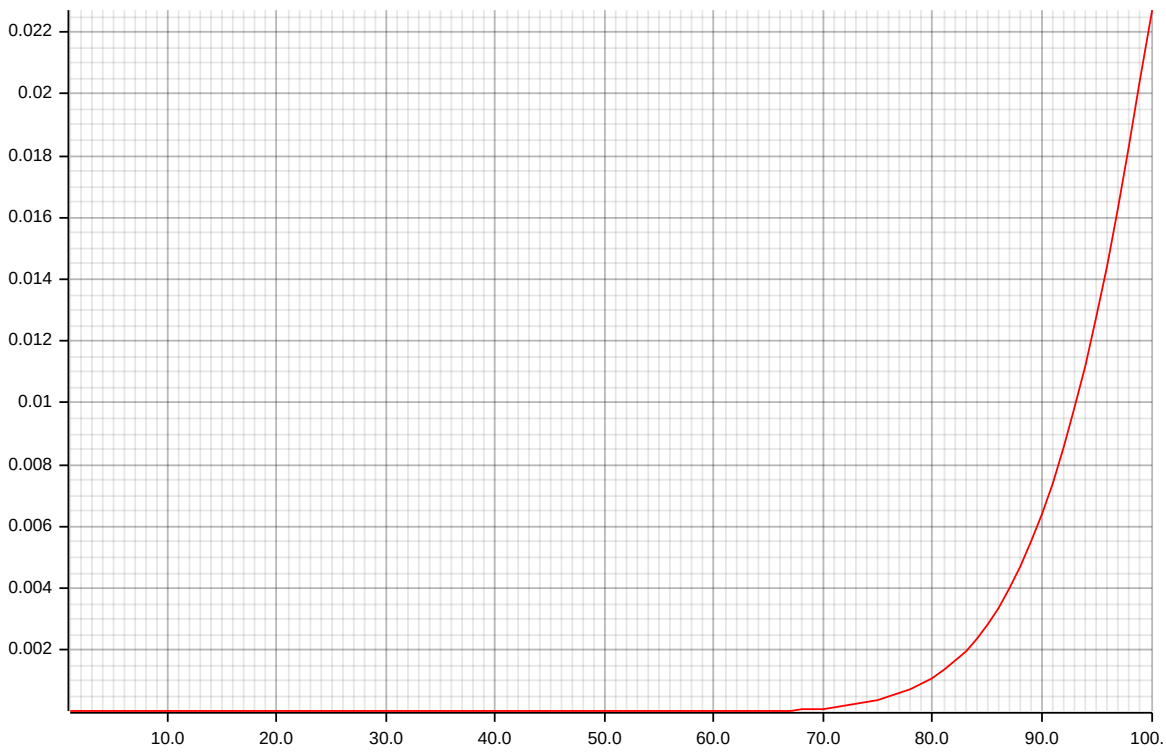
Сначала делаем это для λ_1.

In [15]:
```
let mut block_prob = vec![];
for lambda1 in 1..=100 {
    let rho1 = lambda1 as f64 / &mu1;
    let dist = stationary_prob_distribution(rho1, rho2, c as u8, g as u8);
    block_prob.push((lambda1 as f32, dist.last().cloned().unwrap() as f32));
}
block_prob
```

Out[15]:
```
[(1.0, 1.77e-43), (2.0, 2.4478834e-37), (3.0, 1.0591664e-33), (4.0, 4.28617
9e-31), (5.0, 4.7081145e-29), (6.0, 2.2570824e-27), (7.0, 6.0856095e-26),
(8.0, 1.0738625e-24), (9.0, 1.3677995e-23), (10.0, 1.3447435e-22), (11.0,
1.0705925e-21), (12.0, 7.151608e-21), (13.0, 4.1184912e-20), (14.0, 2.08837
33e-19), (15.0, 9.482566e-19), (16.0, 3.9085558e-18), (17.0, 1.4789452e-1
7), (18.0, 5.1855205e-17), (19.0, 1.6980775e-16), (20.0, 5.2282743e-16), (2
1.0, 1.5222797e-15), (22.0, 4.212393e-15), (23.0, 1.112621e-14), (24.0, 2.8
158101e-14), (25.0, 6.851038e-14), (26.0, 1.6073124e-13), (27.0, 3.6457583e
-13), (28.0, 8.014026e-13), (29.0, 1.7108645e-12), (30.0, 3.5540039e-12),
(31.0, 7.196365e-12), (32.0, 1.4226139e-11), (33.0, 2.7495598e-11), (34.0,
5.20248e-11), (35.0, 9.64826e-11), (36.0, 1.7557211e-10), (37.0, 3.1381067e
-10), (38.0, 5.5142735e-10), (39.0, 9.534293e-10), (40.0, 1.6233456e-9), (4
1.0, 2.723784e-9), (42.0, 4.5067874e-9), (43.0, 7.3581092e-9), (44.0, 1.186
1042e-8), (45.0, 1.888738e-8), (46.0, 2.9725653e-8), (47.0, 4.6259967e-8),
(48.0, 7.12169e-8), (49.0, 1.0850298e-7), (50.0, 1.6366135e-7), (51.0, 2.44
48414e-7), (52.0, 3.618256e-7), (53.0, 5.306725e-7), (54.0, 7.7153976e-7),
(55.0, 1.1122747e-6), (56.0, 1.5903685e-6), (57.0, 2.2558906e-6), (58.0, 3.
175183e-6), (59.0, 4.4354697e-6), (60.0, 6.150551e-6), (61.0, 8.467788e-6),
(62.0, 1.1576572e-5), (63.0, 1.5718519e-5), (64.0, 2.1199596e-5), (65.0, 2.
8404434e-5), (66.0, 3.781301e-5), (67.0, 5.00199e-5), (68.0, 6.5756234e-5),
(69.0, 8.591444e-5), (70.0, 0.00011157569), (71.0, 0.00014404001), (72.0,
0.0001848587), (73.0, 0.0002358686), (74.0, 0.0002992276), (75.0, 0.0003774
505), (76.0, 0.0004734441), (77.0, 0.00059054035), (78.0, 0.0007325257), (7
9.0, 0.0009036656), (80.0, 0.0011087212), (81.0, 0.0013529578), (82.0, 0.00
16421421), (83.0, 0.0019825266), (84.0, 0.002380821), (85.0, 0.0028441472),
(86.0, 0.00337998), (87.0, 0.003996072), (88.0, 0.0047003627), (89.0, 0.005
500877), (90.0, 0.0064056087), (91.0, 0.007422402), (92.0, 0.008558822), (9
3.0, 0.009822028), (94.0, 0.011218651), (95.0, 0.012754676), (96.0, 0.01443
534), (97.0, 0.016265037), (98.0, 0.01824725), (99.0, 0.020384496), (100.0,
0.022678297)]
```

In [16]:
```
draw_chart(&block_prob, "lambda1 vs E")
```

Out[16]:



После этого -- для λ_2.

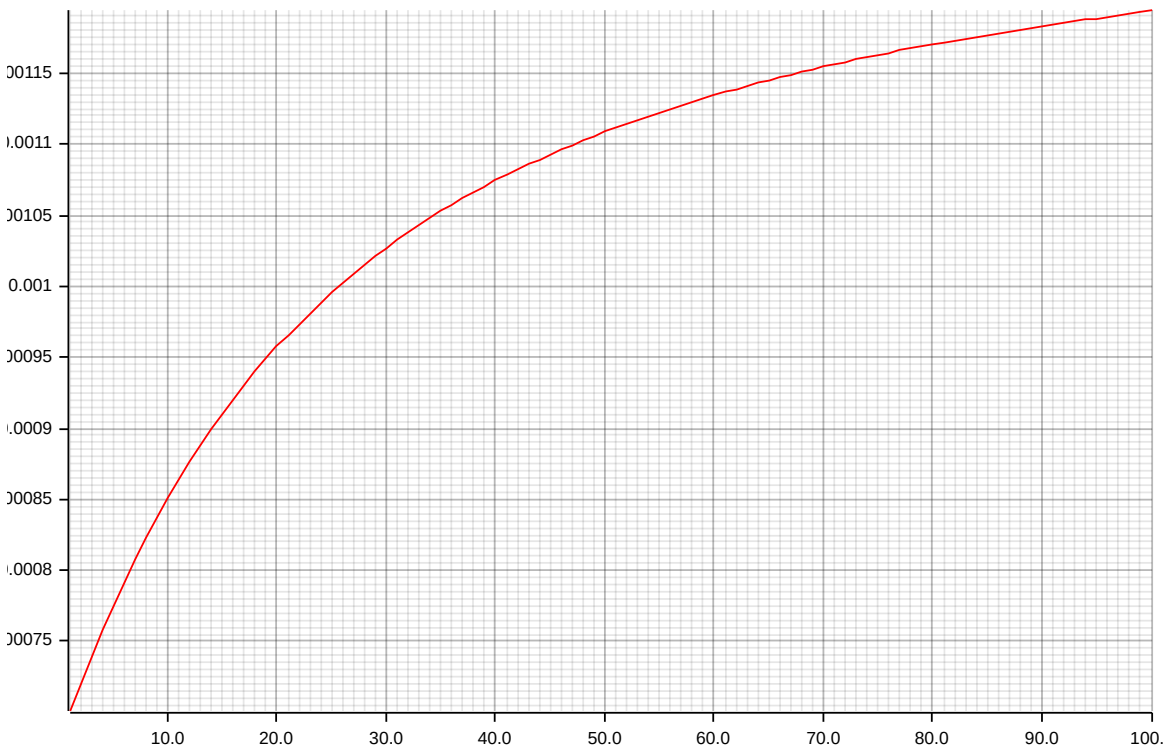
In [17]:

```
let mut block_prob = vec![];
for lambda2 in 1..=100 {
    let rho2 = lambda2 as f64 / &mu1;
    let dist = stationary_prob_distribution(rho1, rho2, c as u8, g as u8);
    block_prob.push((lambda2 as f32, dist.last().cloned().unwrap() as f32));
}
block_prob
```

Out[17]:
```
[(1.0, 0.0007004019), (2.0, 0.0007204047), (3.0, 0.0007394996), (4.0, 0.000
75771636), (5.0, 0.0007750877), (6.0, 0.00079164817), (7.0, 0.0008074333),
(8.0, 0.0008224789), (9.0, 0.0008368207), (10.0, 0.0008504938), (11.0, 0.00
086353236), (12.0, 0.0008759696), (13.0, 0.0008878374), (14.0, 0.000899166
3), (15.0, 0.00090998545), (16.0, 0.0009203226), (17.0, 0.00093020406), (1
8.0, 0.00093965477), (19.0, 0.00094869814), (20.0, 0.00095735653), (21.0,
0.0009656507), (22.0, 0.0009736004), (23.0, 0.0009812242), (24.0, 0.0009885
395), (25.0, 0.0009955628), (26.0, 0.0010023093), (27.0, 0.0010087935), (2
8.0, 0.001015029), (29.0, 0.0010210287), (30.0, 0.0010268046), (31.0, 0.001
0323678), (32.0, 0.0010377292), (33.0, 0.0010428985), (34.0, 0.0010478852),
(35.0, 0.0010526981), (36.0, 0.0010573457), (37.0, 0.0010618357), (38.0, 0.
0010661754), (39.0, 0.0010703718), (40.0, 0.0010744317), (41.0, 0.001078361
1), (42.0, 0.0010821657), (43.0, 0.0010858513), (44.0, 0.0010894231), (45.
0, 0.001092886), (46.0, 0.0010962446), (47.0, 0.0010995032), (48.0, 0.00110
26664), (49.0, 0.0011057378), (50.0, 0.0011087212), (51.0, 0.0011116203),
(52.0, 0.0011144385), (53.0, 0.0011171789), (54.0, 0.0011198446), (55.0, 0.
0011224386), (56.0, 0.0011249635), (57.0, 0.001127422), (58.0, 0.001129816
6), (59.0, 0.0011321497), (60.0, 0.0011344235), (61.0, 0.0011366402), (62.
0, 0.0011388018), (63.0, 0.0011409104), (64.0, 0.0011429678), (65.0, 0.0011
449758), (66.0, 0.001146936), (67.0, 0.0011488502), (68.0, 0.0011507199),
(69.0, 0.0011525466), (70.0, 0.0011543317), (71.0, 0.0011560766), (72.0, 0.
0011577826), (73.0, 0.001159451), (74.0, 0.0011610829), (75.0, 0.001162679
5), (76.0, 0.0011642419), (77.0, 0.0011657713), (78.0, 0.0011672686), (79.
0, 0.0011687347), (80.0, 0.0011701707), (81.0, 0.0011715774), (82.0, 0.0011
729557), (83.0, 0.0011743064), (84.0, 0.0011756304), (85.0, 0.0011769284),
(86.0, 0.0011782012), (87.0, 0.0011794494), (88.0, 0.0011806737), (89.0, 0.
0011818749), (90.0, 0.0011830536), (91.0, 0.0011842104), (92.0, 0.001185345
8), (93.0, 0.0011864605), (94.0, 0.001187555), (95.0, 0.0011886299), (96.0,
0.0011896857), (97.0, 0.0011907227), (98.0, 0.0011917417), (99.0, 0.0011927
43), (100.0, 0.001193727)]
```

In [18]:
```
draw_chart(&block_prob, "lambda2 vs E")
```

Out[18]:



Для того, чтобы нарисовать график среднего числа обслуживаемых запросов, нужно сделать то же самое: по разным значениям λ посчитать распределение вероятности, из него -- среднее количество, и отобразить для него точку. Сначала делаем это, варьируя λ_1, а затем λ_2.

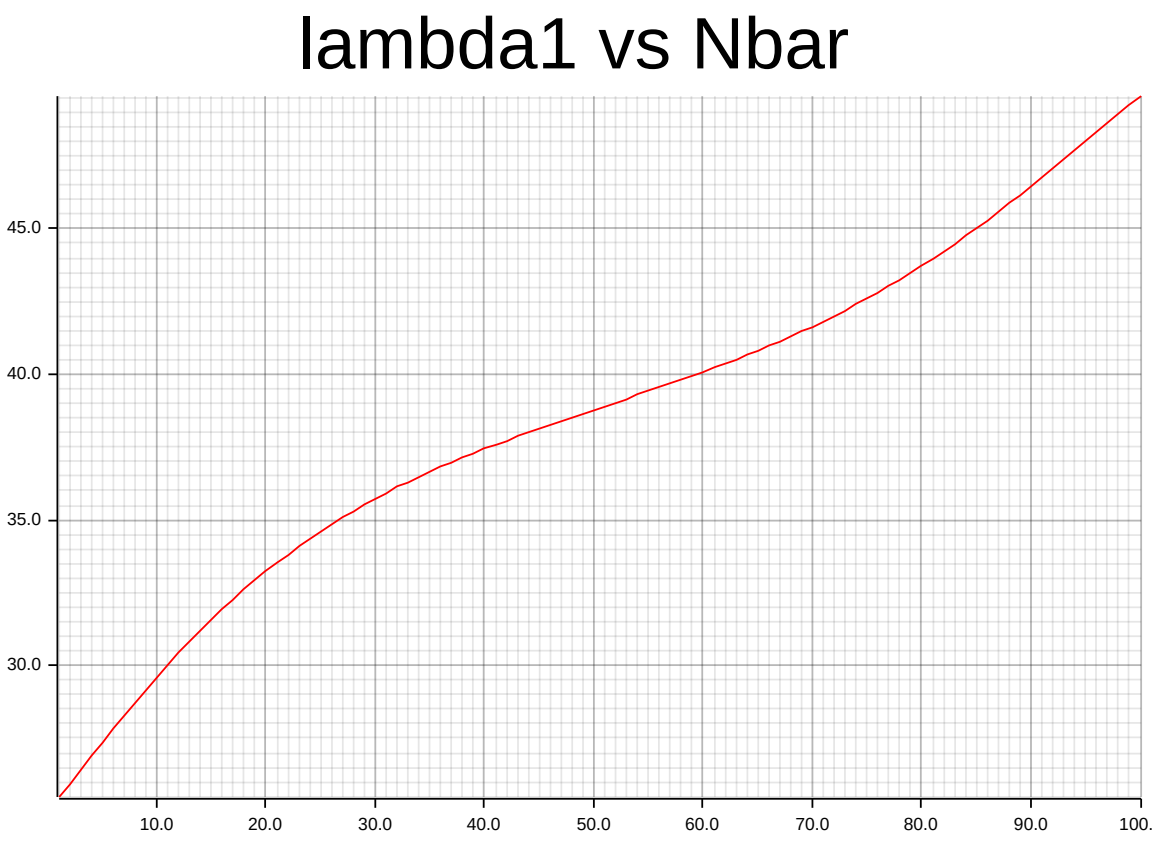
In [19]:

```
let mut avg_req_counts = vec![];
for lambda1 in 1..=100 {
    let rho1 = (lambda1 as f64) / &mu1;
    let dist = stationary_prob_distribution(rho1, rho2, c as u8, g as u8);
    let avg: f64 = dist.iter().enumerate().map(|(i,v)| i as f64 * v).sum();
    avg_req_counts.push((lambda1 as f32, avg as f32));
}
avg_req_counts
```

Out[19]:
```
[(1.0, 25.452143), (2.0, 25.93603), (3.0, 26.415709), (4.0, 26.890446), (5.
0, 27.35946), (6.0, 27.821955), (7.0, 28.277117), (8.0, 28.724144), (9.0, 2
9.162264), (10.0, 29.590746), (11.0, 30.00892), (12.0, 30.416199), (13.0, 3
0.812069), (14.0, 31.196117), (15.0, 31.568027), (16.0, 31.927584), (17.0,
32.274662), (18.0, 32.60924), (19.0, 32.93138), (20.0, 33.241222), (21.0, 3
3.538982), (22.0, 33.824936), (23.0, 34.09942), (24.0, 34.3628), (25.0, 34.
615498), (26.0, 34.85795), (27.0, 35.09061), (28.0, 35.313957), (29.0, 35.5
28473), (30.0, 35.734642), (31.0, 35.932945), (32.0, 36.123863), (33.0, 36.
307865), (34.0, 36.485416), (35.0, 36.656967), (36.0, 36.822956), (37.0, 3
6.983807), (38.0, 37.139935), (39.0, 37.291744), (40.0, 37.439613), (41.0,
37.583927), (42.0, 37.72504), (43.0, 37.863308), (44.0, 37.99907), (45.0, 3
8.13266), (46.0, 38.264397), (47.0, 38.394592), (48.0, 38.523563), (49.0, 3
8.6516), (50.0, 38.779007), (51.0, 38.90607), (52.0, 39.03308), (53.0, 39.1
60324), (54.0, 39.288082), (55.0, 39.41664), (56.0, 39.54628), (57.0, 39.67
7288), (58.0, 39.809944), (59.0, 39.94454), (60.0, 40.081352), (61.0, 40.22
068), (62.0, 40.36281), (63.0, 40.508038), (64.0, 40.65666), (65.0, 40.8089
68), (66.0, 40.965263), (67.0, 41.12584), (68.0, 41.290997), (69.0, 41.4610
18), (70.0, 41.636196), (71.0, 41.816803), (72.0, 42.00311), (73.0, 42.1953
7), (74.0, 42.393818), (75.0, 42.598667), (76.0, 42.810104), (77.0, 43.0282
9), (78.0, 43.253345), (79.0, 43.485355), (80.0, 43.72435), (81.0, 43.9703
2), (82.0, 44.2232), (83.0, 44.482857), (84.0, 44.749104), (85.0, 45.02168
7), (86.0, 45.300285), (87.0, 45.584522), (88.0, 45.87394), (89.0, 46.1680
3), (90.0, 46.466225), (91.0, 46.767902), (92.0, 47.072395), (93.0, 47.37
9), (94.0, 47.686993), (95.0, 47.995613), (96.0, 48.304104), (97.0, 48.6117
1), (98.0, 48.917686), (99.0, 49.22131), (100.0, 49.521885)]
```

In [20]:
```
draw_chart(&avg_req_counts, "lambda1 vs Nbar")
```

Out[20]:

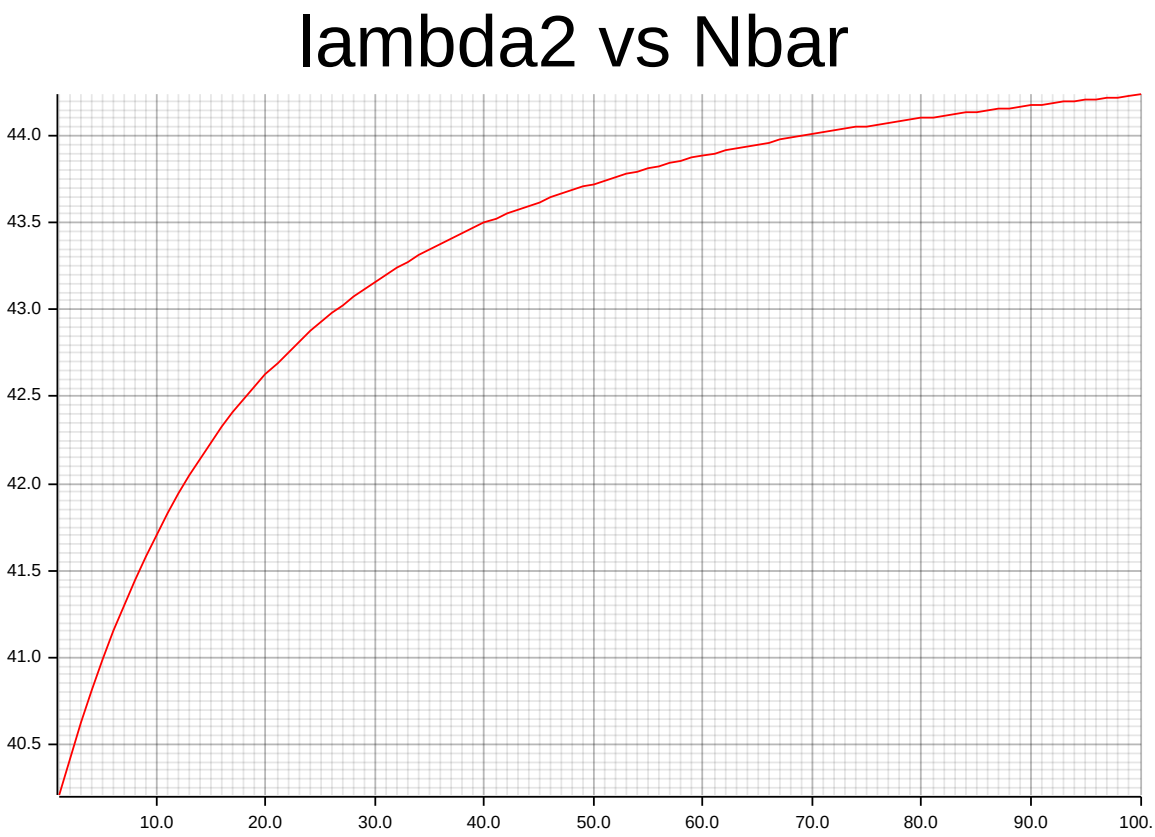


In [21]:
```rust
let mut avg_req_counts = vec![];
for lambda2 in 1..=100 {
    let rho2 = (lambda2 as f64) / &mu2;
    let dist = stationary_prob_distribution(rho1, rho2, c as u8, g as u8);
    let avg: f64 = dist.iter().enumerate().map(|(i,v)| i as f64 * v).sum();
    avg_req_counts.push((lambda2 as f32, avg as f32));
}
avg_req_counts
```

Out[21]:
```
[(1.0, 40.20374), (2.0, 40.419373), (3.0, 40.62076), (4.0, 40.80891), (5.0,
40.984753), (6.0, 41.14919), (7.0, 41.303047), (8.0, 41.44711), (9.0, 41.58
2104), (10.0, 41.7087), (11.0, 41.82752), (12.0, 41.93914), (13.0, 42.04409
4), (14.0, 42.142864), (15.0, 42.23591), (16.0, 42.32364), (17.0, 42.40643
7), (18.0, 42.48465), (19.0, 42.558605), (20.0, 42.6286), (21.0, 42.69490
4), (22.0, 42.757774), (23.0, 42.817436), (24.0, 42.874107), (25.0, 42.9279
82), (26.0, 42.979248), (27.0, 43.028065), (28.0, 43.074593), (29.0, 43.118
973), (30.0, 43.16134), (31.0, 43.201813), (32.0, 43.24051), (33.0, 43.2775
3), (34.0, 43.31298), (35.0, 43.346947), (36.0, 43.379513), (37.0, 43.4107
6), (38.0, 43.440758), (39.0, 43.469578), (40.0, 43.49728), (41.0, 43.5239
3), (42.0, 43.54958), (43.0, 43.57428), (44.0, 43.59808), (45.0, 43.62102
5), (46.0, 43.64316), (47.0, 43.664524), (48.0, 43.685154), (49.0, 43.70508
6), (50.0, 43.72435), (51.0, 43.74298), (52.0, 43.76101), (53.0, 43.77845
8), (54.0, 43.795357), (55.0, 43.811726), (56.0, 43.82759), (57.0, 43.8429
8), (58.0, 43.857903), (59.0, 43.872383), (60.0, 43.886444), (61.0, 43.900
1), (62.0, 43.913364), (63.0, 43.926258), (64.0, 43.938793), (65.0, 43.9509
85), (66.0, 43.962845), (67.0, 43.97439), (68.0, 43.985626), (69.0, 43.9965
7), (70.0, 44.007233), (71.0, 44.01762), (72.0, 44.027744), (73.0, 44.0376
2), (74.0, 44.04725), (75.0, 44.056644), (76.0, 44.06581), (77.0, 44.0747
6), (78.0, 44.083496), (79.0, 44.09203), (80.0, 44.100365), (81.0, 44.1085
1), (82.0, 44.11647), (83.0, 44.124252), (84.0, 44.131863), (85.0, 44.13930
5), (86.0, 44.146584), (87.0, 44.153706), (88.0, 44.16068), (89.0, 44.16750
3), (90.0, 44.174187), (91.0, 44.18073), (92.0, 44.18714), (93.0, 44.1934
2), (94.0, 44.199574), (95.0, 44.205605), (96.0, 44.211517), (97.0, 44.2173
1), (98.0, 44.222996), (99.0, 44.228573), (100.0, 44.23404)]
```

In [22]:
```rust
draw_chart(&avg_req_counts, "lambda2 vs Nbar")
```

Out[22]:


lambda2 vs Nbar
44.0
43.5
43.0
42.5
42.0
41.5
41.0
40.5
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
100.

In [ ]: